Chapter 1

Isosurfaces

Ross Whitaker School of Computing
University of Utah
Salt Lake City, Utah 84112

1.1 Introduction

1.1.1 Motivation

This chapter addresses mechanisms for analyzing and processing volumes in a way that deals specifically with *isosurfaces.* The underlying philosophy is to use isosurfaces as a modeling technology that can serve as an alternative to parameterized models is several important applications in visualization and computer graphics. This chapter presents the mathmatics and numerical techniques for describing the geometry of isosurfaces and manipulating their shapes in prescribed ways. This chapter starts with a basic introduction into the notation and fundamental concepts and then presents the geometry of isosurfaces. It describes the method of level sets, i.e., moving isosurfaces, and presents the mathmatical and numerical methods they entail. It then shows some application examples and describes *VISPACK*, a *C++*, object-oriented library the performs volume processing and level-set modeling.

1.1.2 Isosurfaces

Modeling Surfaces With Volumes

When considering surface models for graphics and visualization, one is faced with a staggering variety of options including meshes, spline-based patches, constructive solid goemetry, implicit blobs, and particle systems. These options can be divided into two basic classes — explicit (parameterized) models and implicit models. With an implicit model, one specifies the model as a *level set* of a scalar function,

$$\phi : \begin{array}{c} U \\ x, y, zr \end{array} \begin{array}{c} \mapsto \mathrm{I\!R} \\ k \end{array}, \tag{1.1}$$

where $U \subset \mathrm{I\!R}^3$ is the domain of the volume (and the *range* of the surface model). Thus, a surface S is

$$S = \{\boldsymbol{x} | \phi(\boldsymbol{x}) = k\} . \tag{1.2}$$

The choice of k is arbitrary, and ϕ is sometimes called the *embedding.* Notice that surfaces defined in this way divide U into a clear inside and outside—such surfaces are allways closed whever they do not intersection the boundary of the domain.

The implicit strategy begs the question of how to represent ϕ. Historically, implicit models are represented using linear combinations of *basis* functions. These basis or potential functions usually have several degrees of

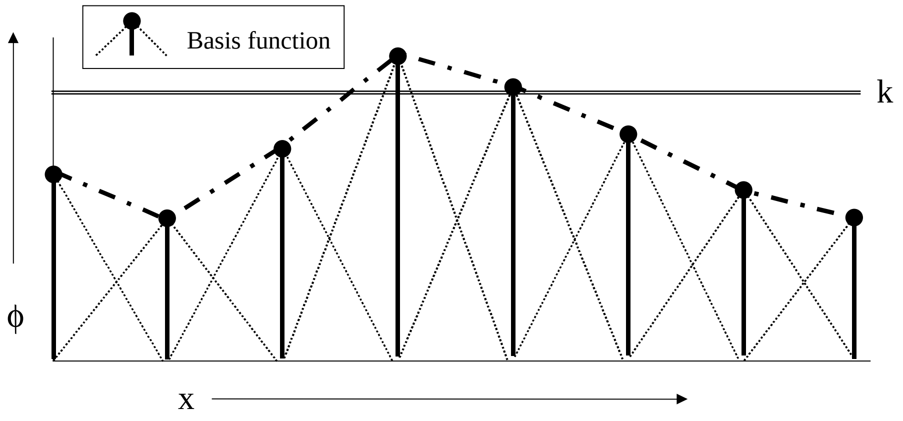


Figure 1.1: A volume can be considered as an implicit model with a large number of local basis functions.

freedom including 3D position, size, and orientation. By combining these functions, one can create complex objects. Typical models might contain several hundred to several thousands of such primitives. This is the strategy behind the "blobby" models proposed by Blinn [1].

While such an implicit modeling strategy offers a variety of new modeling tools, it has some limitations. In particular, the global nature of the potential functions limits one's ability to model *local* surface deformations. Consider a point $\boldsymbol{x} \in \mathcal{S}$ where $\mathcal{S}$ is the level surface associated with a model $\phi = \sum_i \alpha_i$, and α_i is one of the individual potential functions that comprise that model. Suppose one wishes to move the surface at the point $\boldsymbol{x}$ in a way that maintains continuity with the surrounding neighborhood. With multiple, global basis functions one must decide which basis function or combination of basis functions to alter and at the same time control the effects on other parts of the surface. The problem is generally ill posed — there are many ways to adjust the basis functions so that $\boldsymbol{x}$ will move in the desired direction and yet it may be impossible to eliminate the effects of those movements on other disjoint parts of the surface. These problems can be overcome, however they usually entail hueristics that tie the behavior of the surface deformation to the choice of representation [2].

An alternative to using a small number of *global* basis functions is to use a relatively large number of *local* basis functions. This is the principle behind using a volume as an implicit model. A volume is a discrete sampling of the embedding ϕ. It is also an implicit model with a very large number of basis functions, as shown in figure 1.1. The total number of basis functions is fixed, as are their positions (grid points) and extent. One can change only the magnitude of each basis function, i.e., each basis function has only one degree of freedom. A typical volume of size $128 \times 128 \times 128$ contains over a million such basis functions. The shape of each basis function is open to interpretation — it depends on how one interpolates the values between the grid points. A trilinear interpolation, for instance, implies a basis function that is a piecewise cubic polynomial with a value of one at the grid point and zero at neighboring grid points. Another advantage of using volumes as implicit models, is that for the purposes of analysis we can treat the volume as a continuous function whose values can be *set* at each point according to the needs. Once the continuous analysis is complete we can map the algorithm into the discrete domain using standard methods of numerical analysis. The sections that follow discuss how to compute the geometry of surfaces that are represented as volumes and how to manipulate the shapes of those surfaces by changing the grey-scale values in the volume.

Isosurface Extraction and Visualization

Depending on the application, a 3D grid of data, i.e. a volume, may not be a suitable model representation. For instance, if the goal is make measurements of an object or visualize its shape, an explicit model might be necessary. A variety of methods exist for extracting parameteric models of isosurfaces from volumes. The most prevalent method is to isolate iso-surface crossings along gridlines in a volume (between voxels along the 3 cardinal directions) and then to link these points together to form triangles and meshes. This is the strategy of marching cubes [3] and other related approaches []. However, extracting a parameteric surface is not essential for visualization, and a variety of direct methods [] are now computationally feasible and arguably superior in quality. This chapter does not address the issue of extracting or rendering isosurfaces, but rather studies the geometry of isosurfaces and how to manipulate them directly by changing the grey-scale values in the underlying volume. Thus, we propose volumes as a mechanism for studying and deforming surfaces, regardless of the ultimate form of the output. Their are many ways of rendering or visualizing them and and these techniques are beyond the scope of this discussion.

1.2 Surface Normals

The surface normal of the iso-surface is given by the normalized gradient vector. Typically, we identify a surface normal with a point in the volume domain D. That is

$$\boldsymbol{n}(\boldsymbol{x}) = \frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|} \text{ where } \boldsymbol{x} \in D. \tag{1.3}$$

The convention regarding the direction of this vector is arbitrary; the negative of the normalized gradient magnitude is also normal to the iso-surface. The gradient vector points towards that side of the iso-surface which has greater values (i.e. brighter). When rendering, the convention is to use *outward pointing* normals, and the sign of the gradient must be adjusted accordingly. However, for most applications any consistent choice of normal vector will suffice. On a discrete grid, one must also decide how to approximate the gradient vector (i.e., first partial derivatives). In many cases centralized differences will suffice. However, in the presence of noise, especially when volume rendering, it is sometimes helpful to compute first deriviatives using some smoothing filter (e.g., convolution with a Gaussian). When using the normal vector to solve certain kinds of partial differential equations, it is sometimes necessary to approximate the gradient vector with discrete, one-sided differences, as discussed in sucessive sections.

1.3 Second-Order Structure

The second-structure of the isosurface can be computed for the first- and second-order structure of the embedding, ϕ. All of the isosurface shape information is contained field of normals given by $\boldsymbol{n}(\boldsymbol{x})$. The 3×3 matrix of derivatives of this vector,

$$N = [\boldsymbol{n}_x \;\; \boldsymbol{n}_y \;\; \boldsymbol{n}_z] \tag{1.4}$$

describes the second-order structure of the surface. In differential geometric terms, the second-order structure is characterized by a quadratic patch that shares first- and second-order contact with the surface at a point (i.e., tangent plane and osculating circles). This *principle directions* of the surface are those of the quadratic, and the *principle curvatures* are the curvatures in those directions.

The *mean curvature* is the mean of the two principle curvatures, which is one half of the trace of $N(\boldsymbol{x})$ [4]:

$$H = \frac{\phi_x^2(\phi_{yy}+\phi_{zz}) + \phi_y^2(\phi_{xx}+\phi_{zz}) + \phi_z^2(\phi_{xx}+\phi_{yy}) - 2\phi_x\phi_y\phi_{xy} - 2\phi_x\phi_z\phi_{xz} - 2\phi_y\phi_z\phi_{yz}}{2(\phi_x^2+\phi_y^2+\phi_z^2)^{3/2}} \tag{1.5}$$

$$\tag{1.6}$$

The *Gaussian curvature* is the product of the principle curvatures:

$$K = \frac{\begin{array}{c}\phi_z^2(\phi_{xx}\phi_{yy} - \phi_{xy}\phi_{xy}) + \phi_y^2(\phi_{xx}\phi_{zz} - \phi_{xz}\phi_{xz}) + \phi_x^2(\phi_{yy}\phi_{zz} - \phi_{yz}\phi_{yz}) \\ + 2(\phi_x\phi_y(\phi_{xz}\phi_{yz} - \phi_{xy}\phi_{zz}) + \phi_x\phi_z(\phi_{xy}\phi_{yz} - \phi_{xz}\phi_{yy}) + \phi_y\phi_z(\phi_{xy}\phi_{xz} - \phi_{yz}\phi_{xx}))\end{array}}{(\phi_x^2 + \phi_y^2 + \phi_z^2)^2}. \tag{1.7}$$

The total curvature, also called the deviation from flatness, D, is the root sum of squares of the two principle curvatures, which can be derived from H and K, i.e.,

$$D^2 = 4H^2 - 2K. \tag{1.8}$$

Notice, these measures exist at every point in U, and at each point they describe the geometry of the particular isosurface that passes through that point. All of these quantities can be computed on a discrete volume using finite differences, as described in successive sections.

1.4 Deformable Surfaces

This section begins with mathematics for describing surface deformations on parametric models. The result is an evolution equation for a surface. Each of the terms in this evolution equation can be re-expressed in a way that is independent of the parameterization. Finally, the evolution equation for a parametric surface gives rise to an evolution equation (differential equation) on a volume, which encodes the shape of that surface as a level set.

1.4.1 Surface Deformation

A regular surface $\mathcal{S} \subset \mathbb{R}^3$ is a collection of points in 3D that can be be represented *locally* as a continuous function. In geometric modeling a surface is typically represented as a two-parameter object in a three-dimensional space, i.e., a surface is local a mapping $\boldsymbol{S}$:

$$\boldsymbol{S} : \underset{r}{V} \times \underset{s}{V} \mapsto \underset{x,y,z}{\mathbb{R}^3}, \tag{1.9}$$

where $V \times V\mathbb{R}^2$, and the bold notation refers specifically to a parameterized surface (vector-valued function). A deformable surface exhibits some motion over time. Thus $\boldsymbol{S} = \boldsymbol{S}(r, s, t)$, where $t \in \mathbb{R}^+$. We assume second-order-continuous, orientable surfaces; therefore at every point on the surface (and in time) there is surface normal $\boldsymbol{N} = \boldsymbol{N}(r, s, t)$. We use $\mathcal{S}_t$ to refer to the entire set of points on the surface.

Local deformations of $\boldsymbol{S}$ can described by an evolution equation, i.e., a differential equation on $\boldsymbol{S}$ that incorporates the position of the surface, local and global shape properties, and responses to other forcing functions. That is,

$$\frac{\partial \boldsymbol{S}}{\partial t} = \boldsymbol{G}((\boldsymbol{S}, \boldsymbol{S}_r, \boldsymbol{S}_s, \boldsymbol{S}_{rr}, \boldsymbol{S}_{rs}, \boldsymbol{S}_{ss}, \ldots), \tag{1.10}$$

where the subscripts represent partial derivatives with respect to those parameters. The evolution of $\boldsymbol{S}$ can be described by a sum of terms that depends on both the geometry of $\boldsymbol{S}$ and the influence of other functions or data.

There are a variety of differential expressions that can be combined for different applications. For instance, the model could move in response to some directional "forcing" function [5, 6], $\boldsymbol{F} : U \mapsto \mathbb{R}^3$, that is

$$\frac{\partial \boldsymbol{S}}{\partial t} = \boldsymbol{F}(\boldsymbol{S}). \tag{1.11}$$

Alternatively, the surface could expand and contract with a spatially-varying speed. For instance,

$$\frac{\partial \boldsymbol{S}}{\partial t} = G(\boldsymbol{S})\boldsymbol{N} \tag{1.12}$$

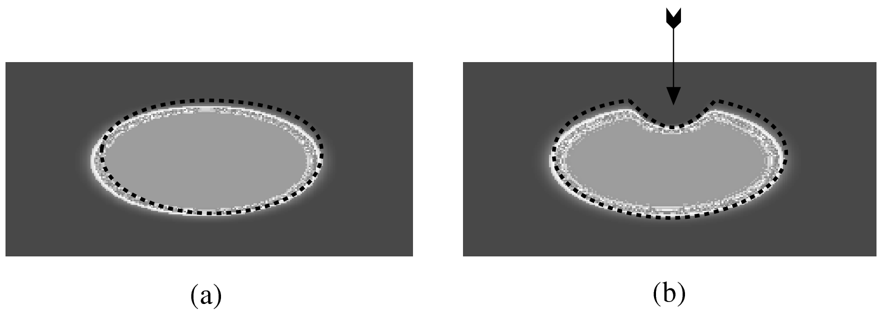


Figure 1.2: Level-set models represent curves and surfaces implicitly using greyscale images: a) an ellipse is represented as the level set of an image, b) to change the shape we modify the greyscale values of the image.

where $G : \mathbb{R}^3 \mapsto \mathbb{R}$ is a signed speed function. The evolution might also depend on the surface geometry itself. For instance,

$$\frac{\partial \boldsymbol{S}}{\partial t} = \boldsymbol{S}_{rr} + \boldsymbol{S}_{ss} \tag{1.13}$$

describes a surface that moves in way that is becomes more *smooth* with respect to its own parameterization. This motion can be combined with the motion of Equation 1.11 to produce a model that is pushed by a forcing function but maintains a certain smoothness in its shape and parameterization. There is a myriad of terms that depend on both the differential geometry of the surface and outside forces or functions to control the evolution of a surface.

1.5 Deformation: The Level Set Approach

The method of level-sets, proposed by Osher and Sethian [7] and described extensively in [4], provides the mathematical and numerical mechanisms for computing surface deformations as time-varying iso-values of ϕ by solving a partial differential equation on the 3D grid. That is, the level-set formulation provides a set of numerical methods that describe how to manipulate the greyscale values in a volume, so that the isosurfaces of ϕ move in a prescribed manner (shown in Figure 1.2). For example, let $\mathrm{d}\boldsymbol{x}/\mathrm{d}t$ be the movement of a point on a surface as it deforms, such that it can be expressed in terms of the position of $\boldsymbol{x} \in U$ and the geometry of the surface at that point, which is, in turn, a differential expression of the implicit function, ϕ. If we assume that the point $\boldsymbol{x}$ remains on the k level set of ϕ as it moves, then the total derivative of $\phi(\boldsymbol{x}(t), t) = k$ is zero, and we have

$$\frac{\partial \phi}{\partial t} = -\nabla\phi \cdot \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t}. \tag{1.14}$$

In the level-set formulation, surface movements can depend only on the position $\boldsymbol{x}$ and the surface shape, which is expressed in terms of the shape of the embedding ϕ. That is

$$\frac{\partial \phi}{\partial t} = -\nabla\phi \cdot \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = -\nabla\phi \cdot \boldsymbol{F}(\boldsymbol{x}, \mathrm{D}\phi, \mathrm{D}^2\phi, \ldots), \tag{1.15}$$

where $D^n\phi$ is the set of order-n derivatives of ϕ evaluated at $\boldsymbol{x}$. Because this relationship applies to every level-set of ϕ, i.e. all values of k, this equation can be applied to all of U, and therefore the movements of *all* the level-set surfaces embedded in ϕ can be calculated from Equation 1.15. Such level-set methods are well documented in the literature [7, 8] for applications such as computational physics, image processing [9, 10], computer vision [11, 12], medical image analysis [13, 12, 14, 15, 10, 16], and 3D reconstruction [17, 18].

The level-set representation has a number of practical and theoretical advantages over conventional surface models, especially in the context of deformation. First, level-set models are topologically flexible, they can easily represent complicated surface shapes that can, in turn, form holes, split to form multiple objects, or merge with other objects to form a single structure. These models can incorporate many (millions) of degrees of freedom, and therefore they can accommodate complex shapes. Indeed, the shapes formed by the

	Effect	**Parametric Evolution**	**Level-Set Evolution**	**Parameter Assumptions**
1	External force	$\boldsymbol{F}$	$\boldsymbol{F} \cdot \nabla\phi$	None
2	Expansion/ contraction	$G(\boldsymbol{x})\boldsymbol{N}$	$G(\boldsymbol{x})\lvert\nabla\phi(\boldsymbol{x},t)\rvert$	None
3	Mean curvature	$S_{rr} + S_{ss}$	$H\lvert\nabla\phi\rvert$	Orthonormal
4	Gauss curvature	$S_{rr} \times S_{ss}$	$K\lvert\nabla\phi\rvert$	Orthonormal
5	Second order	S_{rr} or S_{ss}	$\left(H \pm \sqrt{H^2 - K}\right)\lvert\nabla\phi\rvert$	Principle curvatures

Table 1.1: A list of evolution terms for parameteric models has a corresponding expression on the embedding, ϕ, associated with the level-set models.

level sets of ϕ are restricted only by the resolution of the sampling. Thus, there is no need to reparameterize the model as it undergoes significant changes in shape.

1.5.1 Deformation Modes

In the case of parametric surfaces, one can choose from a variety of different expressions to construct an evolution equation that is appropriate for a particular application. For each of those parametric expressions, there is a corresponding expression that can be formulated on ϕ, the volume in which the level-set models are embedded. In constructing evolutions on levels sets, there can be no reference to the underlying surface parameterization (terms depending on r and s in Equations 1.9 through 1.13). This has two important implications: 1) only those surface movements that are normal to the surface are represented (any other movement is the equivalent of a reparameterization) 2) all of the derivatives with respect to surface parameters r and s must be expressed in terms of invariant surface properties that can be derived without a parameterization.

Consider the term $\boldsymbol{S}_{rr} + \boldsymbol{S}_{ss}$ from equation 1.13. If r, s is an orthonormal parameterization, the effect of that term is based purely on surface shape, not on the parameterization, and the expression $\boldsymbol{S}_{rr}+\boldsymbol{S}_{ss}$ is twice the *mean curvature*, H, of the surface. The corresponding level-set formulation is given by Equation 1.5.

Table 1 shows a list of expressions used in the evolution of parameterized surfaces and their equivalents for level-set representations. Also given are the assumptions about the parameterization that give rise to the level-set expressions.

1.6 Numerical Methods

By taking the strategy of embedding surface models in volumes, we have converted equations that describe the movement of surface points to nonlinear, partial differential equations defined on a volume, which is generally a rectilinear grid. The expression $u^n_{i,j,k}$ refers to the nth time step at position i, j, k, which has an associated value in the 3D domain of the continuous volume $\phi(x_i, y_j, z_k)$. The goal is to solve the differential equation consisting of term from Tabletab:terms on the discrete grid $u^n_{i,j,k}$.

The discretization of these equations raises two important issues. First is the availability of accurate, stable numerical schemes for solving these equations. Second is the problem of computational complexity and the fact that we have converted a *surface* problem to a *volume* problem, increasing the dimensionality of the domain over which the evolution equations must be solved.

The level-set terms in Table 1 are combined, based on the needs of the application, to create a partial differential equation on $\phi(\boldsymbol{x}, t)$. The solutions to these equations are computed using finite differences. Along the time axis solutions are obtained using finite *forward* differences, begining with an initial model (i.e., volume) and stepping sequentially through a series of discrete times steps (which are denoted as superscripts on u). Thus the update equation is:

$$u^{n+1}_{i,j,k} = u^n_{i,j,k} + \Delta t \Delta u^n_{i,j,k}, \tag{1.16}$$

The term $\Delta u^n_{i,j,k}$ is a discrete approximation to $\partial\phi/\partial t$, which consists of a weighted sum of terms such as those in Table 1.5.1. Those terms must, in turn, be approximated using finite differences on the volume grid.

1.6.1 Up-wind Schemes

The terms in Table 1 fall into two basic categories: the first-order terms (items 1 and 2 in Table 1) and the second-order terms (items 3 through 5). The first-order terms describe a moving wave front with a space-varying velocity (expression 1) or speed (expression 2). Equations of this form cannot be solved with a simple finite forward difference scheme. Such schemes tend to overshoot, and they are unstable. To address this issue Osher and Sethian [19] have proposed an *up-wind* scheme. The up-wind method relies on a one-sided derivative that looks in the up-wind direction of the moving wavefront, and thereby avoids the over-shooting associated with finite forward differences.

We denote the type of descrete difference using supersripts on a difference operator, i.e., $\delta^{(+)}$ for forward differences, $\delta^{(-)}$ for backward differences, and δ for centralized differences. For instance, differences in the x direction on a discrete grid, $u_{i,j,k}$, with domain X and uniform spacing h are defined as

$$\begin{aligned}
\delta^{(+)}_x u_{i,j,k} &\triangleq (u_{i+1,j,k} - u_{i,j,k})/h, & (1.17)\\
\delta^{(-)}_x u_{i,j,k} &\triangleq (u_{i,j,k} - u_{i-1,j,k})/h, \text{ and} & (1.18)\\
\delta_x u_{i,j,k} &\triangleq (u_{i+h,j,k} - u_{i-h,j,k})/(2h), & (1.19)\\
& & (1.20)
\end{aligned}$$

where we have left off the time superscript for conciseness. Second-order terms are computed using the *tightest-fitting* centralized difference operators. For example,

$$\begin{aligned}
\delta_{xx} u_{i,j,k} &\triangleq \left(u_{i+1,j,k} + u_{i-1,j,k} - 2u_{i,j,k}\right)/h^2, & (1.21)\\
\delta_{zz} u_{i,j,k} &\triangleq \left(u_{i,j,k+1} + u_{i,j,k-1} - 2u_{i,j,k}\right)/h^2, \text{ and} & (1.22)\\
\delta_{xy} u_{i,j,k} &\triangleq \delta_x \delta_y u_{i,j,k} & (1.23)\\
& & (1.24)
\end{aligned}$$

The discrete approximation to the first-order terms of in Table 1.5.1 are computed using the up-wind proposed by Osher and Sethian [7]. This strategy avoids overshooting by approximating the gradient of ϕ using a one-sided differences in the direction that is up-wind of the moving level-set thereby ensuring that no *new* contours are created in the process of updating $u^n_{i,j,k}$ (as depicted in Figure 1.3). The scheme is separable along each axis (i.e., x, y, and z).

Consider Term 1 in Table 1.5.1. If we use superscripts to denote the vector components, i.e., $\boldsymbol{F}(x,y,z) = (F^{(x)}(x,y,z), F^{(y)}(x,y,z), F^{(z)}(x,y,z))$, the up-wind calulation for a grid point $u^n_{i,j,k}$ is

$$\boldsymbol{F}(x_i,y_i,z_i)\cdot\nabla\phi(x_i,y_j,z_k,t) \approx \sum_{q\in\{x,y,z\}} F^{(q)}(x_i,y_i,z_i)\begin{cases} \delta^+_q u^n_{i,j,k} & F^{(q)}(x_i,y_i,z_i) > 0\\ \delta^i_q u^n_{i,j,k} & F^{(q)}(x_i,y_i,z_i) < 0 \end{cases} \tag{1.25}$$

The time steps are limited—the fastest moving wave front can move only one grid unit per iteration. That is

$$\Delta t_{\boldsymbol{F}} \leq \frac{1}{\sum_{q\in\{x,y,z\}} \sup_{i,j,k\in X}\{|\nabla F^{(q)}(x_i,y_j,z_k)|\}}. \tag{1.26}$$

For Term 2 in Table 1.5.1 the direction of the moving surface depends on the normal, and therefore the same up-wind strategy is applied in a slightly different form.

$$G(x_i,y_j,z_k)|\nabla\phi(x_i,y_j,z_k,t)| \approx \sum_{q\in\{x,y,z\}} G(x_i,y_i,z_i)\begin{cases} \max^2(\delta^+_q u^n_{i,j,k},0) + \min^2(\delta^-_q u^n_{i,j,k},0) & G(x_i,y_i,z_i) > 0\\ \min^2(\delta^+_q u^n_{i,j,k},0) + \max^2(\delta^-_q u^n_{i,j,k},0) & G(q)(x_i,y_i,z_i) < 0 \end{cases} \tag{1.27}$$

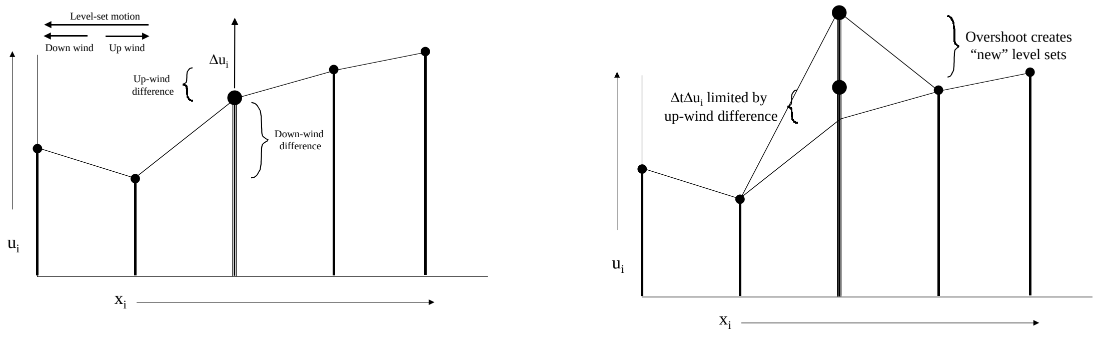


Figure 1.3: The up-wind numerical scheme uses one-sided derivatives to prevent overshooting and the creation of new level sets.

The time steps are, again, limited by the fastest moving wave front:

$$\Delta t_G \leq \frac{1}{3 \sup_{i,j,k \in X}\{|\nabla G(x_i, y_j, z_k)|\}} \tag{1.28}$$

To compute approximation the update to the second-order terms in Table 1.5.1 requires only centralized differences . Thus, the mean curvature is approximated as:

$$\begin{aligned} H^n_{i,j,k} \quad = \quad & \frac{1}{2}\left(\left(\delta_x u^n_{i,j,k}\right)^2 + \left(\delta_y u^n_{i,j,k}\right)^2 + \left(\delta_z u^n_{i,j,k}\right)^2\right)^{-1}\left[\left(\left(\delta_y u^n_{i,j,k}\right)^2 + \left(\delta_z u^n_{i,j,k}\right)^2\right)\delta_{xx} u^n_{i,j,k}\right. \\ & + \left(\left(\delta_z u^n_{i,j,k}\right)^2 + \left(\delta_x u^n_{i,j,k}\right)^2\right)\delta_{yy} u^n_{i,j,k} + \left(\left(\delta_x u^n_{i,j,k}\right)^2 + \left(\delta_y u^n_{i,j,k}\right)^2\right)\delta_{zz} u^n_{i,j,k} \\ & \left. -2\delta_x u^n_{i,j,k}\delta_y u^n_{i,j,k}\delta_{xy} u^n_{i,j,k} - 2\delta_y u^n_{i,j,k}\delta_z u^n_{i,j,k}\delta_{yz} u^n_{i,j,k} - 2\delta_z u^n_{i,j,k}\delta_x u^n_{i,j,k}\delta_{zx} u^n_{i,j,k}\right] \end{aligned} \tag{1.29}$$

The time steps are limited, for stability, to

$$\Delta t_H \leq \frac{1}{6}. \tag{1.30}$$

When combining terms, the maximum time steps for each terms as scaled by one over the weighting coefficient for that term.

1.6.2 Narrow-Band Methods

If one is interested in only *a single level set*, the formulation described previously is not efficient. This is because solutions are usually computed over the entire domain of ϕ. The solutions, $\phi(x, y, z, t)$ describe the evolution of an embedded family of contours. While this dense family of solutions might be advantageous for certain applications, there are other applications that require only a single surface model. In such applications the calculation of solutions over a dense field is an unnecessary computational burden, and the presence of contour families can be a nuisance because further processing might be required to extract the level set that is of interest.

Fortunately, the evolution of a single level set, $\phi(\boldsymbol{x}, t) = k$, is not affected by the choice of embedding. The evolution of the level sets is such that they evolve independently (to within the error introduced by the discrete grid). Furthermore, the evolution of ϕ is important only in the vicinity of that level set. Thus, one should perform calculations for the evolution of ϕ only in a neighborhood of the surface $\mathcal{S} = \{\boldsymbol{x}|\phi(\boldsymbol{x}) = k\}$. In the discrete setting, there is a particular subset of grid points whose values control a particular level set

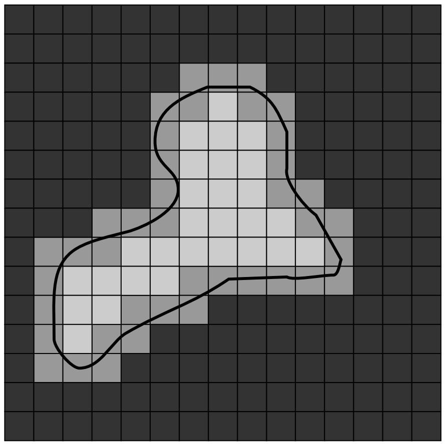


Figure 1.4: A level curve of a 2D scalar field passes through a finite set of cells. Only those grid points nearest to the level curve are relevant to the evolution of that curve.

(see Figure 1.4). Of course, as the surface moves, that subset of gridpoints must change to account for its new position.

Adalsteinson and Sethian [20] propose a *narrow-band* approach which follows this line of reasoning. The narrow-band technique constructs an embedding of the evolving curve or surface via a signed distance transform. The distance transform is truncated, i.e, computed over a finite width of only m points that lie within a specified distance to the level set. The remaining points are set to constant values to indicate that they do not lie within the narrow band, or *tube* as they call it. The evolution of the surface (they demonstrate it for curves in the plane) is computed by calculating the evolution of u only on the set of grid points that are within a fixed distance to the initial level set, i.e. within the narrow band. When the evolving level set approaches the edge of the band (see Figure 1.5), they calculate a new distance transform and a new embedding, and they repeat the process. This algorithm relies on the fact that the embedding is not a critical aspect of the evolution of the level set. That is, the embedding can be transformed or recomputed at any point in time, so long as such a transformation does not change the position of the kth level set, and the evolution will be unaffected by this change in the embedding.

Despite the improvements in computation time, the narrow-band approach is not optimal for several reasons. First it requires a band of significant width ($m = 12$ in the examples of [20]) where one would like to have a band that is only as wide as necessary to calculate the derivatives of u near the level set (e.g. $m = 2$). The wider band is necessary because the narrow-band algorithm trades off two competing computational costs. One is the cost of stopping the evolution and computing the position of the curve and distance transform (to sub-cell accuracy) and determining the domain of the band. The other is the cost of computing the evolution process over the entire band. The narrow-band method also requires additional techniques, such as smoothing, to maintain the stability at the boundaries of the band, where some grid points are undergoing the evolution and nearby neighbors are stationary.

1.6.3 The Sparse-Field Method

The basic premise of the narrow band algorithm is that computing the distance transform is so costly that it cannot be done at every iteration of the evolution process. The strategy proposed here is to use an approximation to the distance transform that makes it feasible to recompute the neighborhood of the level-set model at each time step. Computation of the evolution equation is computed on a band of grid points that is only on point wide. The embedding is extended from the active points to a neighborhood around those points that is precisely the width needed at each time. This extension is done via a fast distance transform approximation.

This approach has several advantages. First, the algorithm does precisely the number of calculations needed to compute the next position of the level curve. It does not require explicitly recalculating the

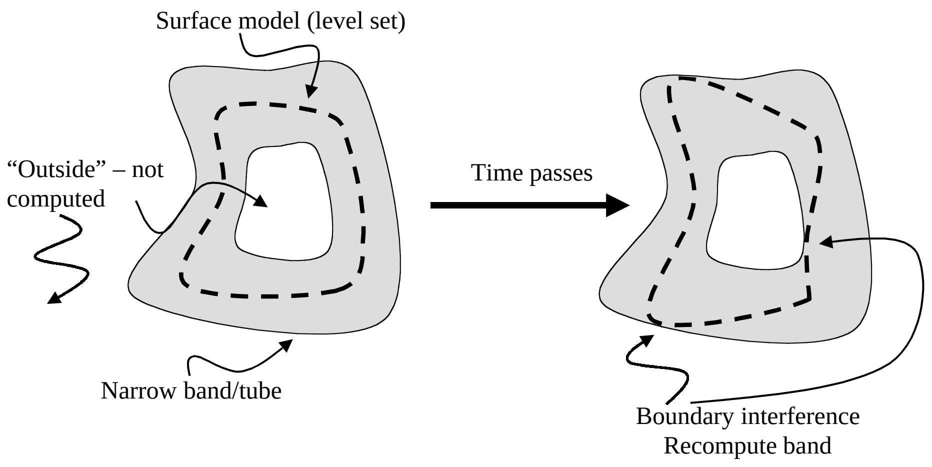


Figure 1.5: The narrow band scheme limits computation to the vicinity of the specific level set. As the level-set moves near the edge of the band the process is stopped and the band recomputed.

positions of level sets and their distance transforms. Because the number of points being computed is so small, it is feasible to use a linked-list to keep track of them. Thus, at each iteration the algorithm visits only those points adjacent to the k-level curve. For large 3D data sets, the very process of incrementing a counter and checking the status of all of the grid points is prohibitive.

The *sparse-field* algorithm is analogous to a locomotive engine that lays down tracks before it and picks them up from behind. In this way the number of computations increases with the size of the model rather than the resolution of the embedding. Also, the sparse-field approach identifies a single level set with a specific set of points whose values control the position of that level set. This allows one to compute external forces to an accuracy that is better than the grid spacing of the model, resulting in a modeling system that is more accurate for various kinds of "model fitting" applications.

The sparse-field algorithm takes advantage of the fact that a k-level surface, S, of a discrete image u (of any dimension) has a set of cells through which it passes, as shown in figure 1.4. The set of grid points adjacent to the level set is called the *active set*, and the individual elements of this set are called *active points*. As a first-order approximation, the distance of the level set from the center of any active point is proportional to the value of u divided the gradient magnitude at that point. Because all of the derivatives (up to second order) in this approach are computed using nearest neighbor differences, only the active points and their neighbors are relevant to the evolution of the level-set at any particular time in the evolution process. The strategy is to compute the evolution given by equation 1.15 on the active set and then update neighborhood around the active set using a fast distance transform. Because active points must be adjacent to the level-set model, their positions lie within a fixed distance to the model. Therefore the values of u for locations in the active set must lie within a certain range. When active-point values move out of this *active range* they are no longer adjacent to the model. They must be removed from the set and other grid points, those whose values are moving into the active range, must be added to take their place. The precise ordering and execution of these operations is important to the operation of the algorithm.

The values of the points in the active set can be updated using the up-wind scheme for first-order terms and centralized differences for the mean-curvature flow, as described in the previous sections. In order to maintain stability, one must update the neighborhoods of active grid points in a way that allows grid points to enter and leave the active set without those changes in status affecting their values. Grid points should be removed from the active set when they are no longer the nearest grid point to the zero crossing. If we assume that the embedding u is a discrete approximation to the distance transform of the model, then the distance of a particular grid point, $x_m = (i, j, k)$, to the level set is given by the value of u at that grid point. If the distance between grid points is defined to be unity, then we should remove a point from the active set when the value of u at that point no longer lies in the interval $[-\frac{1}{2}, \frac{1}{2}]$ (see figure 1.6). If the neighbors of that point maintain their distance of 1, then those neighbors will move into the active range just x_m is ready to be removed.

There are two operations that are significant to the evolution of the active set. First, the values of u at active points change from one iteration to the next. Second, as the values of active points pass out of the active range they are removed from the active set and other, neighboring grid points are added to the active set to take their place. The appendix of this paper gives some formal definitions of active sets and the operations that affect them, and it shows that active sets will always form a boundary between positive and negative regions in the image, even as control of the level set passes from one set off active points to another.

Because grid points that are near the active set are kept at a fixed value difference from the active points, active points serve to control the behavior of nonactive grid points to which they are adjacent. The neighborhoods of the active set are defined in *layers*, $L_{+1}, \ldots L_{+N}$ and $L_{-1}, \ldots L_{-N}$, where the i indicates the distance (city block distance) from the nearest active grid point, and negative numbers are used for the outside layers. For notational convenience the active set is denoted L_0.

The number of layers should coincide with the size of the footprint or neighborhood used to calculate derivatives. In this way, the inside and outside grid points undergo no changes in their values that affect or distort the evolution of the zero set. The work in this paper uses surface curvature, which requires only second-order derivatives of ϕ. Second-order derivatives are calculated using a $3 \times 3 \times 3$ kernel (city-block distance 2 to the corners). Therefore only five layers are necessary (2 inside layers, 2 outside layers, and the active set). These layers are denoted L_1, L_2, L_{-1}, L_{-2}, and L_0.

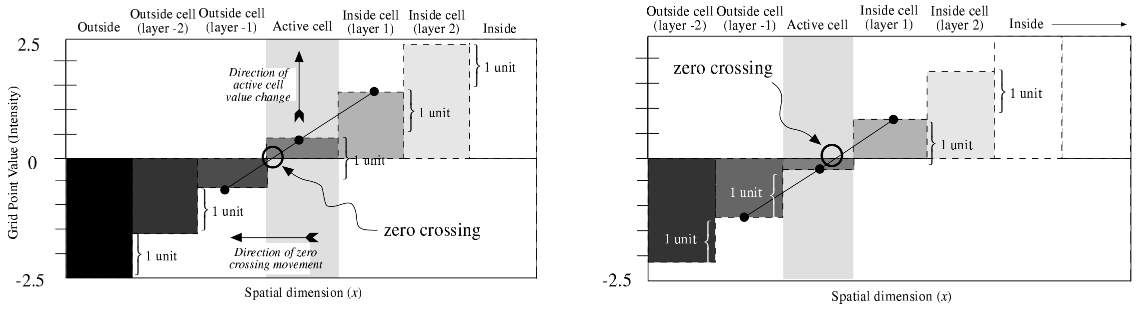


Figure 1.6: The status of grid points and their values at two different points in time show that as the zero crossing moves, *activity* is passed one grid point to another.

The active set has grid point values in the range $[-\frac{1}{2}, \frac{1}{2}]$. The values of the grid points in each neighborhood layer are kept 1 unit from the next layer closest to the active set (as in figure 1.6). Thus the values of layer L_i fall in the interval $[i - \frac{1}{2}, i + \frac{1}{2}]$. For $2N + 1$ layers, the values of the grid points that are totally inside and outside are $N + \frac{1}{2}$ and $-N - \frac{1}{2}$, respectively. The procedure for updating the image and the active set based on surface movements is as follows:

1. For each active grid point, $x_m = (i, j, k)$, do the following:
   (a) Calculate the local geometry of the level set.
   (b) Compute the net change of u_{x_m}, based on the internal and external forces, using some stable (e.g., up-wind) numerical scheme where necessary.
2. For each active grid point x_j add the change to the grid point value and decide if the new value $u^{n+1}_{x_m}$ falls outside the $[-\frac{1}{2}, \frac{1}{2}]$ interval. If so, put x_m on lists of grid points that are changing status, called the *status list*; S_1 or S_{-1}, for $u^{n+1}_{x_m} > 1$ or $u^{n+1}_{x_m} < -1$, respectively.
3. Visit the grid points in the layers L_i in the order $i = \pm 1, \ldots \pm N$, and update the grid point values based on the values (by adding or subtracting one unit) of the next inner layer, $L_{i\mp 1}$. If more than one $L_{i\mp 1}$ neighbor exists then use the neighbor that indicates a level curve closest to that grid point, i.e., use the maximum for the outside layers and minimum for the inside layers. If a grid point in layer L_i has no $L_{i\mp 1}$ neighbors, then it gets demoted to $L_{i\pm 1}$, the next level away from the active set.
4. For each status list $S_{\pm 1}, S_{\pm 2}, \ldots, S_{\pm N}$ do the following:
   (a) For each element x_j on the status list S_i, remove x_j from the list $L_{i\mp 1}$, and add it to the L_i list, or, in the case of $i = \pm(N + 1)$, remove it from all lists.
   (b) Add all $L_{i\mp 1}$ neighbors to the $S_{i\pm 1}$ list.

This algorithm can be implemented efficiently using linked-list data structures combined with arrays to store the values of the grid points and their states as shown in figure 1.7. This requires only those grid points whose values are changing, the active points and their neighbors, to be visited at each time step. The computation time grows as m^{n-1}, where m is the number of grid points along one dimension of u (sometimes called the resolution of the discrete sampling). Computation time for dense-field approach increases as m^n. The m^{n-1} growth in computation time for the sparse-field models is consistent with conventional (parameterized) models, for which computation times increase with the resolution of the domain, rather than the range.

Another important aspect of the performance of the sparse-field algorithm is the larger time steps that are possible. The time steps are limited by the speed of the "fastest" moving level curve, i.e., the maximum

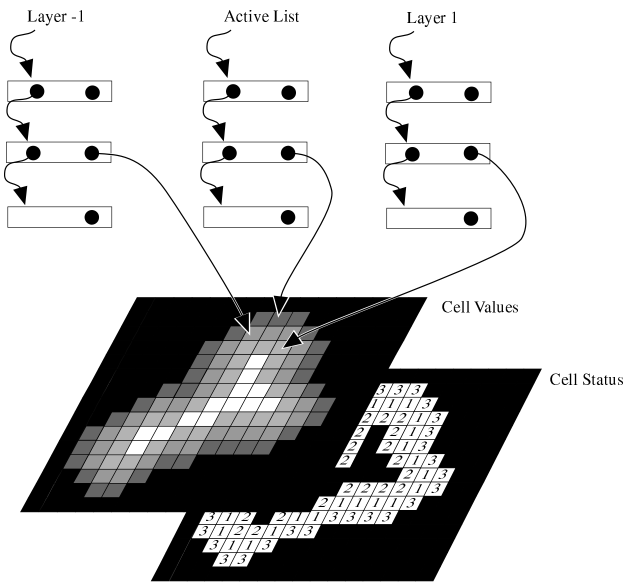


Figure 1.7: Linked-list data structures provide efficient access to those grid points with values and status that must be updated.

of the force function. Because the sparse-field method calculates the movement of level sets over a subset of the image, time steps are bounded from below by those of the dense-field case, i.e.,

$$\sup_{x \in \mathcal{A} \subset X} (g(x)) \leq \sup_{x \in X} (g(x)), \tag{1.31}$$

where $g(x)$ is the space varying speed function and $\mathcal{A}$ is the active set.

Results from previous work [18] have demonstrated several important aspects of the sparse-field algorithm. First, the manipulations of the active set and surrounding layers allow the active set to "track" the deformable surface as it moves. The active set allways divides the inside and outside of the objects it desribes (i.e., it stays closed). Empirical results show significant increases in performance relative to both the computation of full domain and the narrow-band method, as proposed in the literature. Empirical results also show that the sparse-field method is about as accurate as both the full, discrete solution, and the narrow-band method. Finally, because the method positions level sets to subvoxel accuracy it avoids aliasing problems and is more accurate then these other methods when it comes to fitting level-set models to other surfaces

1.7 Applications

This section describes several examples of how level-set surface models can be used to address problems in graphics, visualization, and computer vision. These examples are a small selection of those available in the literature. All of these examples where implemented using the sparse-field algorithm and the VISPack library, which is described in the section that follows.

1.7.1 Surface Morphing

The *morphing* of 3D surfaces is the process of constructing a series of 3D models that constitute a smooth transition from one shape to another (i.e., a homotopy). Such a capability is interesting for creating animations and as a tool for geometric modeling. There is not yet a single, general method for generating such transitional shapes. However, there are several desirable aspects of morphing algorithms that allow us to compare the adequacy of different approaches to surface morphing. Several desirable properties of 3D surface morphing are:

1. The transition process should begin with an *initial* surface and end with a specified *target* surface.
2. The morphing algorithm should apply to a wide range of shapes and topologies.
3. Intermediate surfaces should undergo continuous 3D transitions (rather than continuity only in the image space).
4. A 3D morphing algorithm should incorporate user input easily but should degrade gracefully without it.
5. Transitional shapes should depend only on the surface geometry of the two input shapes and user input.

These requirements are not exhaustive, but they capture many of the practical aspects of 3D morphing.

In this section we show how level-set models provide an algorithm for 3D morphing which meets most of these criteria and compare favorably with existing algorithms. Furthermore, this algorithm is a natural extension of the mathematical principles discussed in previous sections. The strategy is to allow a free-form deformation of one surface (called the *initial* surface) using the signed distance transform of a second surface (the *target* surface). This free-form deformation is combined with an underlying coordinate transformation that gives either a rough global alignment of the two surfaces, or one-to-one relationships between a finite set of landmarks on both the initial and target surfaces. The coordinate transformation can be computed automatically or using user input (as in [21]).

Much of the previous 3D morphing work has focused on morphing parametric models [22, 23] and applies to only very limited classes of shapes and topologies. Several authors have described volumetric techniques. Hughes [24] demonstrates how volumes can provide topological flexibility in surface morphing. Lerios et al. [21] followed up with a volume-based scheme which incorporates user input via underlying coordinate transformations (a known generalization the image warping technique that is often used in image morphing). Neither of these approaches have dealt with the deeper issue of deforming the level sets of a volume, but rather rely on the properties of the embedding. Payne and Toga [25] as well as Cohen-Or et al. [26] fix the embedding problem by using a signed distance transform to create volumes from surfaces. However, interpolating distance transforms can introduce artifacts that violate the properties described in the previous paragraphs, and both of these methods use a discrete distance transform which introduces volume aliasing.

Free-Form Deformations

The distance transform gives the nearest Euclidean distance to a set of points, curve, or surface. For closed surfaces in 3D, the signed distance transform gives a positive distance for points inside and negative for points outside (one can also choose the opposite sign convention).

If two connected shapes overlap then the initial surface can expand or contract using the distance transform of the target. The steady state of such a deformation process is a shape consisting of the zero set of the distance transform of the target. That is, the initial object becomes the target. This is the basis of our 3D morphing algorithm.

Let $D(\boldsymbol{x})$ be the signed distance transform of the target surface, B, and let A be the initial surface. The evolution process which takes a model S from A to B is defined by

$$\frac{\partial \boldsymbol{x}}{\partial t} = \boldsymbol{N}\, D(\boldsymbol{x}), \tag{1.32}$$

where $\boldsymbol{x}(t) \in \mathcal{S}_t$ and $\mathcal{S}_{t=0} = A$. The free-form deformations can be combined with an underlying coordinate transformation. The strategy is to use a coordinate transformation (for instance a translation and rotation) to position the two surfaces near each other. These transformations can capture gross similarities in shape as well as user input. A coordinate transformation is given by

$$\boldsymbol{x}' = T(\boldsymbol{x}, \alpha), \tag{1.33}$$

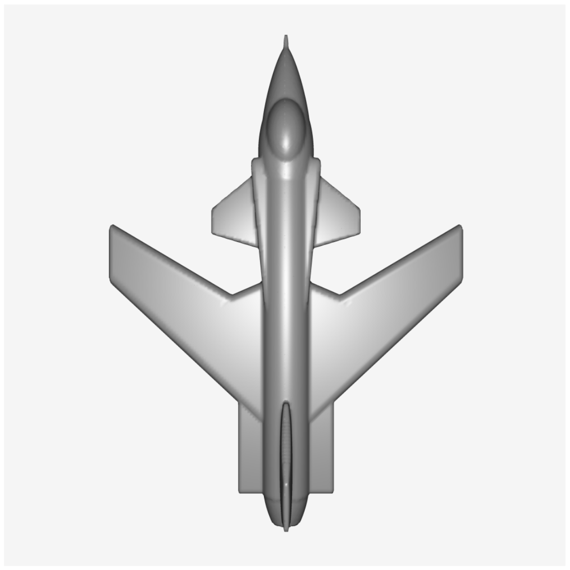


Figure 1.8: A 3D model of a jet that was built using Clockworks, a CSG modeling system.

where $0 \leq \alpha \leq 1$ parameterizes a continuous family of these transformations that begins with identity, i.e. $\boldsymbol{x} = T(\boldsymbol{x}, 0)$. The evolution equation for a parametric surface is

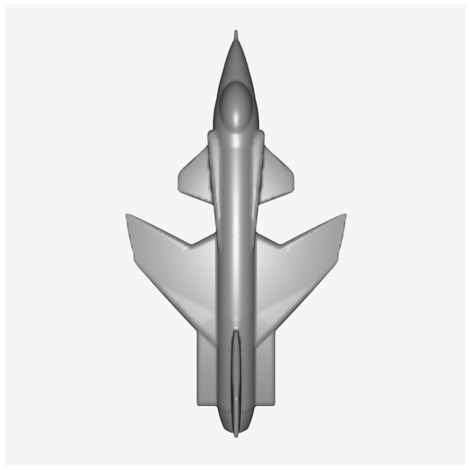
$$\frac{\partial \boldsymbol{x}}{\partial t} = \boldsymbol{N}\ D(T(\boldsymbol{x}, 1)), \tag{1.34}$$

and the corresponding level-set equation is

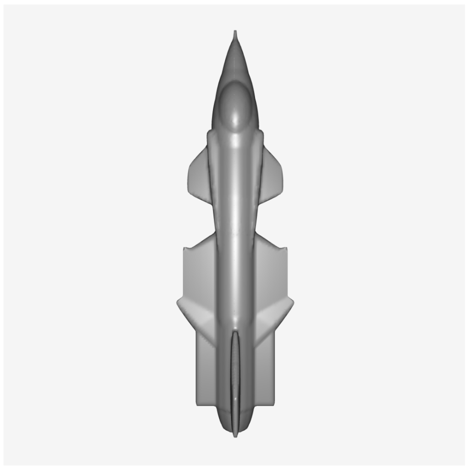
$$\frac{\partial \Phi(\boldsymbol{x}, t)}{\partial t} = |\nabla \Phi(\boldsymbol{x}, t)|\ D(T(\boldsymbol{x}, 1)). \tag{1.35}$$

This process produces a series of transition shapes (parameterized by t). The coordinate transformation can be a global rotation, translation, or scaling, or it might be a *warping of the underlying 3D space* as was used by [21]. Incorporating user input is important for any surface morphing technique, because in many cases finding the best set of transition surfaces depends on context. Only users can apply semantic considerations to the transformation of one object to another. Our assertion, however, is that this underlying coordinate transformation can achieve only some finite similarity between the "warped" initial model and the target, and even this may require a great deal of user input. In the event that a user is not able or willing to define every important correspondence between two objects, some other method must "fill in" the gaps remaining between the initial and target surface. In [21] they propose alpha blending to achieve that smooth transition—really just a fading from one surface to the other. We are proposing the use of the free-form deformations, implemented with level-set models, to achieve a continuous transition between the shapes that result from the underlying coordinate transformation. We have also experimented with ways of automatically orienting and scaling objects, using 3D moments, in order to achieve a significant correspondence between two objects.

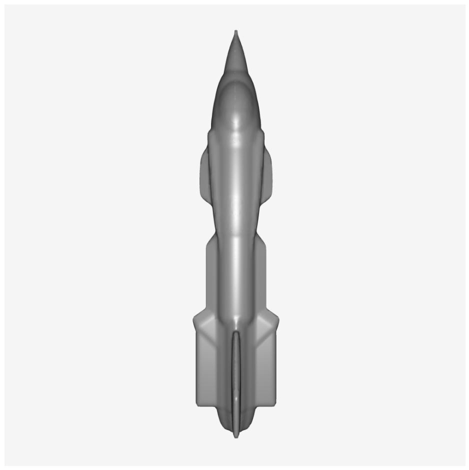
Figure 1.8 shows a 3D model of a jet that was built using Clockworks [27], a CSG modeling system. Lerios et al. [21] demonstrate the transition of a jet to a dart, which was accomplished using 37 user-defined correspondences, roughly a hundred user-defined parameters. Figure 1.9 shows the use of level-set models to construct a set of transition surfaces between a jet and a dart. The triangle mesh is extracted from the volume using marching cubes [3]. These results are obtained without any user input. Distance transforms on the CSG models are computed near the level surface using an analytical description and extended into the volume using a level-set method [28]. User input, in the form of nonlinear coordinate transformations, is easily incorporated into this algorithm, and would allow users to control morphs based on a deeper understanding of the objects involved.



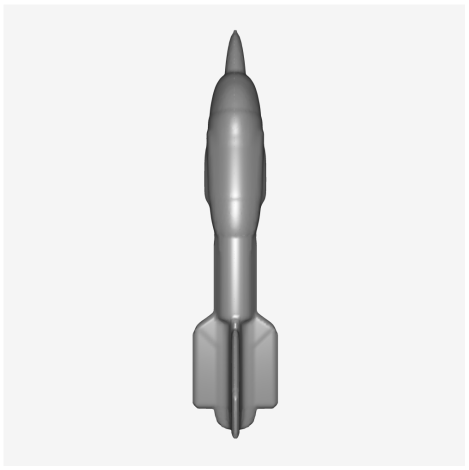
(a)



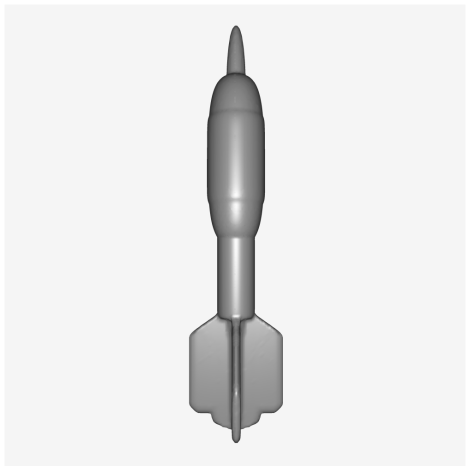
(b)



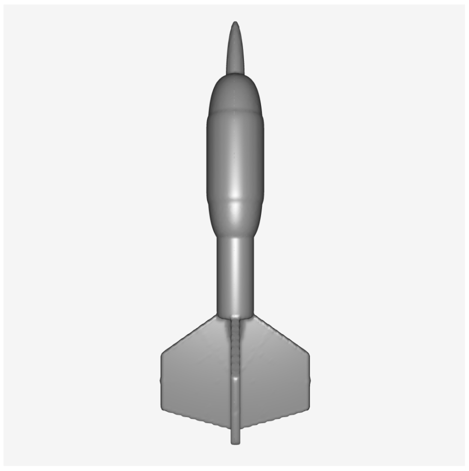
(c)



(d)



(e)



(f)

Figure 1.9: The deformation of the jet to a dart using a level-set model moving with a speed defined by the signed distance transform of the target object.

The application in this section shows how level-set models moving according to the first-order term given in expression 2 in Table 1 can "fit" other objects by moving with a speed that depends on the signed distance transform of the target object. The application in the next section relies on expression 5 of Table 1, a second-order flow that depends on the principle curvatures of the surface itself.

1.7.2 Filleting and Blending Solid Objects

The construction of blending surfaces is an important tool in solid modeling. Geometric solid primitives and their intersections often produce sharp corners or creases that are often not consistent with the real-world objects that they are intended to represent. This section shows how blending can be described as a deformation process, where surfaces move under a geometric flow that can add or remove material based on local curvature information. The result is a method for solid object blending that does not depend on any particular model representation. Thus this method is not restricted to a specific class of shapes or topologies. Additionally, the results are invariant; they do not depend on arbitrary choices of coordinate systems or bases. The only requirement is that the blended objects must be closed surfaces with some known inside-outside function.

Surface blending techniques are typically tied very closely to the choice of geometric primitives. For instance, Middleditch and Sears [29] propose a set-theoretic method for blending solids which relies on low-order algebraic primitives. A fillet at the joint of two tori requires the solution of a degree 32 polynomial. Bloomenthal and Shoemake [30] propose a modeling system based on convolutions, which relies on a skeletonized representation of objects. In general the use of convolution to achieve deformations on implicit shapes results in shapes that reflect both the shape of the model and the embedding, Φ.

The blending method proposed in this section implements an interative smoothing scheme that smooths only along the level set; the final result is independent of the embedding. Consider the case of fillets. We propose that a fillet can be constructed from a process of "filling in" material in places of high curvature. The curvature of a level-set model can be calculated from the embedding, and the deformation of the level set is well defined by the curvature terms in Table 1.

The strategy is to construct a curvature term, k_p, that consists of only positive curvatures. [1] The principle curvatures of the level sets of Φ are functions of Φ and its derivatives. For a specific Φ the principle curvatures are functions of 3-space $k_1(\boldsymbol{x})$ and $k_2(\boldsymbol{x})$. For *adding* material the joint between two objects, we consider only the positive curvature components, i.e.,

$$\frac{\partial \Phi}{\partial t} = |\nabla \Phi| k_p = |\nabla \Phi| k_1^+ + |\nabla \Phi| k_2^+, \tag{1.36}$$

where k^+ consists of only the positive parts of k and is defined as zero elsewhere. Because the use of separate curvature terms can cause over-shooting, the up-wind scheme (treating k_p as a space-varying velocity in the normal direction) is used for this evolution.

Figure 1.10 shows how the positive-curvature flow can be used to construct fillets. No knowledge of the underlying models is necessary. The fillets grow larger as more time passes. The physical extent or position of the fillet can be controlled by either specifying a region of action or by placing a small blob of deformable material in the joint that requires a fillet. Figure 1.11 shows how such a blending capability can be useful in animation. In this case a pair of superquadrics undergo a rigid transformation that controls their relative positions. Level-set models with a positive-curvature flow are used to create a smooth joint between these two primitives. Notice that the positive curvature method does not suffer from the growth or expansion artifacts that are often associated with distance-based blending methods [31].

Thus, a second-order flow can create smooth blends between objects in a way that does not require specific knowledge of the shapes or topologies of the object involved. The application in the next section, 3D scene reconstruction, shows how a combination of first-order and second-order terms from Table 1 are combined to create technique that fits models to data while maintaining certain smoothness constraints and thereby offsetting the effects of noise.

[1]The sign of curvature is defined by the direction of the normals— in this work normals point into the volume enclosed by the object.

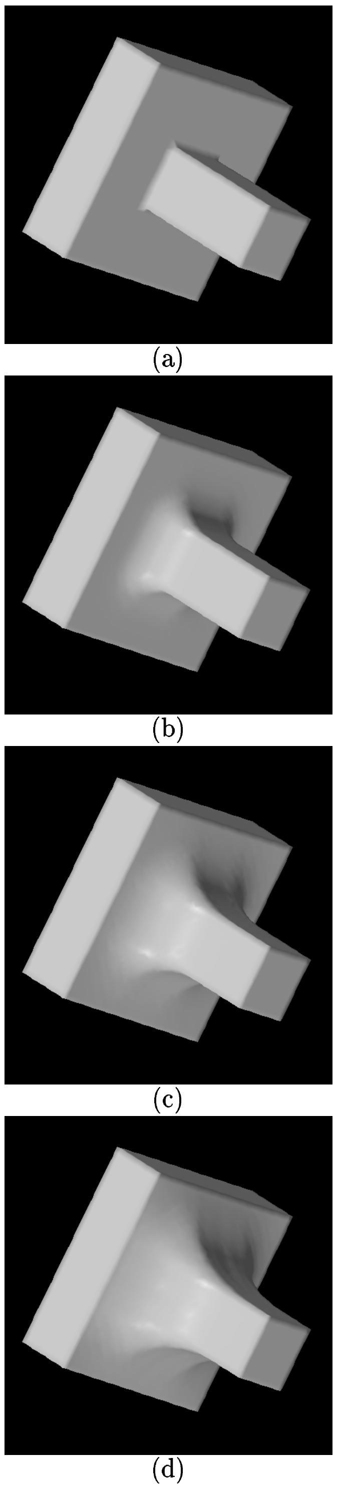


Figure 1.10: Two rectangular solid models are joined by a volumetric fillet that is created from a positive curvature flow.

1.7.3 3D Reconstruction from Multiple Range Maps

Level-set models are useful for problems related to 3D reconstruction. Previous work has presented level-set results derived from noisy 3D data such as MRI [13] and ultrasound [32]. In [33] we have shown how the reconstruction of objects from multiple range maps can be formulated as a problem of finding the surface that optimizes the posterior probability given a set of measurements (noisy range maps) and some information about the a-priori likelihood of different kinds of surfaces. That optimization problem can be expressed as a volume integral which can be solved with level-set models. This section presents the mathematical expressions that result from those formulations and presents some new results: the reconstruction of entire scenes by fitting level-set models to the data from a scanning LADAR (laser ranging and detection) system.

A *range map* is a collection of range measurements taken along different directions (lines of sight) but from a single point of view. Range maps could come from any number of different sources including laser scanners, structured light depth systems, shape from stereo, or shape from motion. We assume that such range maps are noisy and uncertain. The goal is to combine a number of range maps from different points of view to create a 3D structure that reflects the collective confidence and depth measures.

Several examples in the literature have applied parametric models to this task. Turk and Levoy [34], for instance, "zip" together triangle meshes in order to construct 3D objects from sequences of range maps from a laser range finder. They perform minor adjustments to the surface position in order account for ambiguity in the range maps. Their approach assumes very little noise in the input, which is reasonable given the high quality of their range maps. Chen and Medioni [35] use a parametric (triangle mesh) model which expands inside a sequence of range maps. Curless and Levoy [36] describe a volume-based technique for combining range data. They use the signed distance transform to encode volume elements with data that represent the averages (with some allowance for outliers) of multiple measurements. Surfaces of objects are the level sets of volumes. Related approaches are given in [37, 38]. Bajaj et. al. [39] use a Delaunay triangulation to impose a topology on a set of unordered 3D points and then fit trivariate Bernstein-Bezier patches—i.e. a higher-order implicit model—to the data. Muraki [2] uses implicit or blobby models to reconstruct objects from range data. The individual blobs are spherically symmetric 3D potentials that are combined linearly so that they blend together. The resulting models, with approximately 400 primitives are quite coarse.

This work differs from previous work in two ways. First, rather than heuristics, our reconstruction strategy is based on a strategy that solves for the optimal surface estimate. This optimal estimate includes information about one's expectations of the likelihood of different surfaces. The result is not a closed-form solution, but an iterative process that seeks to fit a level-set model to the data while enforcing a kind of smoothness on the data.

Objective function for multiple range maps

The evolution equation for the estimation of optimal surfaces is shown in [33] to consist of two parts:

$$\frac{\partial \boldsymbol{x}}{\partial t} = -G(\boldsymbol{x})\boldsymbol{N} + \rho(\mathcal{S}). \tag{1.37}$$

This first part, $-G(\boldsymbol{x})\boldsymbol{N}$, is the data term, which is a movement with variable speed (as in expression 2 from Table 1) that is the cumulative effect from all of the individual range maps. The second part is the prior, which describes the likelihood of the surface independent of the data. The data term is

$$G(\boldsymbol{x}) = \sum_j c^{(j)}(\boldsymbol{x})\; D^{(j)}(\boldsymbol{x})\; \omega\left(D^{(i)}(\boldsymbol{x})\right)\; \gamma^{(j)}(\boldsymbol{x}), \tag{1.38}$$

where D_j is the signed distance along the line of sight from a range measurement in range map j associated passing through $\boldsymbol{x}$. The function $\omega : \mathbb{R} \mapsto \mathbb{R}$ is a windowing function that limits the penalty of any one range measurement, and $c(\cdot)$ is a confidence function, which is inversely proportional to the level of noise in the range measurement associated with the same line of sight. The term $\gamma(\cdot)$ is an integration constant that takes into account the curvilinear coordinate system of the range scanner.

Thus, a set of range maps creates a scalar function of 3D, which describes the movement of a surface model as it seeks the optimal surface position. In the absence of a prior, $\rho = 0$, the zero set of this function

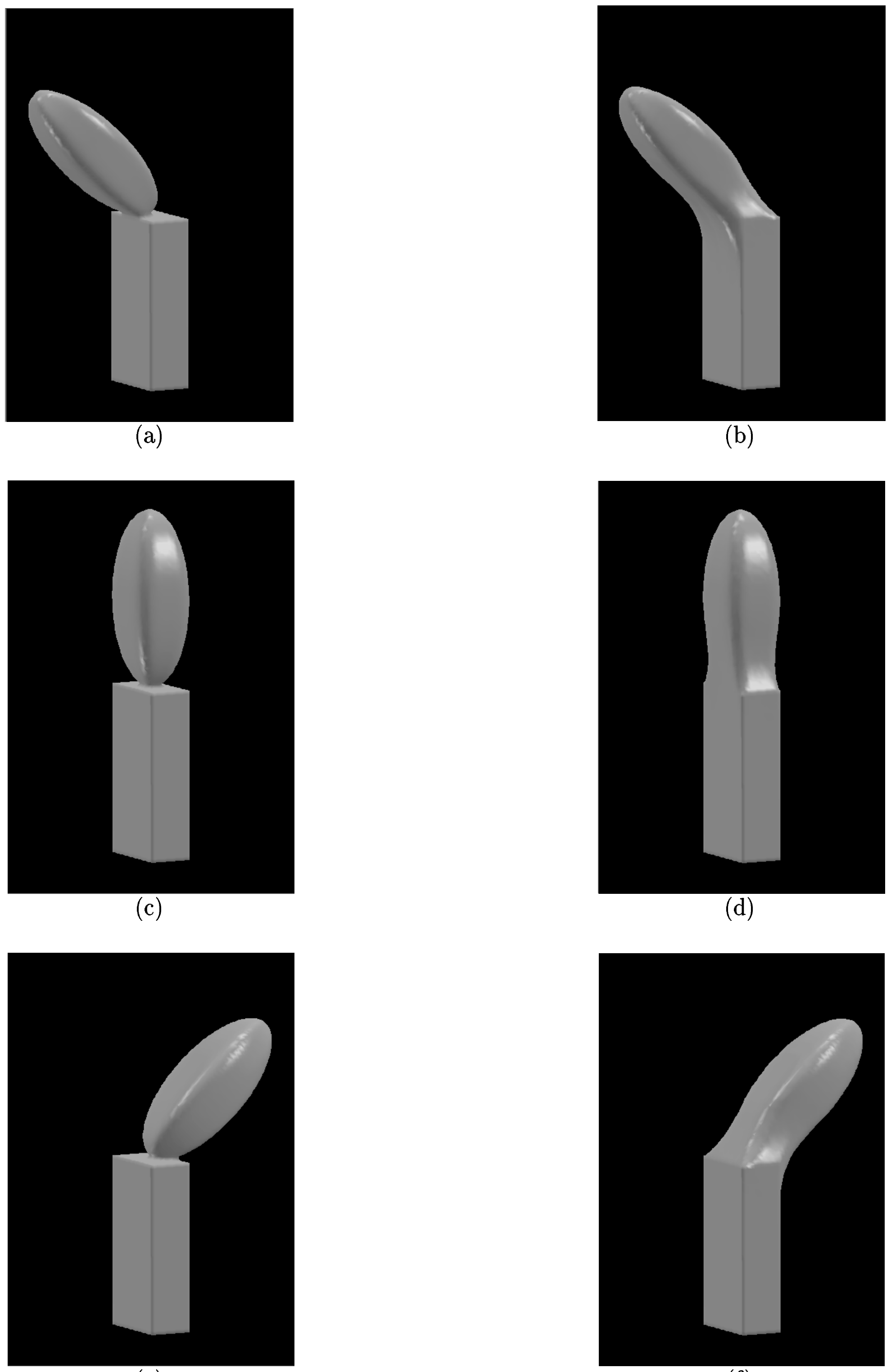


Figure 1.11: A short animation is created by specifying the relative motion between two superquadric components of an object. A positive-curvature flow (applied frame by frame to the joint between the two 3D models) creates a smooth, flexible object.

is the final position (steady state) of that evolving surface. Thus, in the absence of a prior, one could sample $g(\boldsymbol{x})$ and obtain an approximation to the optimal surface estimate. This strategy results in an algorithm that is very much like that of [36].

There are several reasons for going to an iterative scheme for finding optimal solutions. First is the use of a prior. In surface reconstruction, even a very low level of noise can degrade the quality of the rendered surfaces in the final result, and in such cases better reconstructions can be obtained by introducing a prior. Second is aliasing. Discretizing $g(\boldsymbol{x})$ and finding the zero crossings will cause aliasing in those places where the transition from positive to negative is particularly steep. A deformable model can place the surface much more precisely. The third reason for going to an iterative scheme is that despite the windowing function $\omega(\boldsymbol{x})$ there is interference between different range maps at places of high curvature. This problem is addressed by introducing a nonlinearity which is solved in an iterative scheme given by equation 1.37. In this work, the solution of the linear problem, the zero set of $g(\boldsymbol{x})$, serves as the initial estimate for the nonlinear, iterative optimization strategy that results from the inclusion of a prior and a nonlinear term that compensates for lack of any explicit model of self occlusions.

Equation 1.37 includes a *prior*, which is a likelihood function on surface shape. A reasonable choice of prior is one that models objects with less surface area as more likely than objects with more surface area. Alternatively, one could say that given a set of surfaces that are near the data, the algorithm should choose a surface that has less area. Often, but not always, this will be the smoother surface. The $\rho(\mathcal{S})$ that results from this prior is the mean curvature. Therefore the evolution of the surface, using the level-set formulation, that seeks to maximize the posterior probability (given a set of range maps and a prior that penalizes surface area) is

$$\begin{aligned}\frac{\partial \Phi(\boldsymbol{x},t)}{\partial t} &= |\nabla \Phi(\boldsymbol{x})| \sum_j \left(D^{(j)}(\boldsymbol{x})\ \omega\left(D^{(i)}(\boldsymbol{x})\right) \right. \\ &\qquad \left. \times\, \gamma^{(j)}(\boldsymbol{x})\ c^{(j)}(\boldsymbol{x}) \frac{\left(\nabla\Phi \cdot \boldsymbol{n}^{(j)}(\boldsymbol{x})\right)^{+}}{\nabla\Phi \cdot \boldsymbol{n}^{(j)}(\boldsymbol{x})} \right) \\ &\qquad + \beta H, \end{aligned} \tag{1.39}$$

where $\boldsymbol{n}^{(j)}(\boldsymbol{x})$ is the line of sight from a range finder to a 3D point, $\boldsymbol{x}$, β is a free parameter that controls the level of smoothing in the model, and H is the expression for the mean curvature given in equation 1.5.

Figure 1.12 shows a pair of simulated range maps constructed from an analytical description of a torus. These 200×200 pixel range maps are corrupted with additive Gaussian noise that has a standard deviation of 20% (as a function of the smaller of the two radii). Six synthetic noise-corrupted viewpoints of a torus are combined to create a level-set reconstruction of a torus. Figure 1.13(a) shows the initial model (80×80×40 voxels) used for fitting a level-set models to the range data. Figure 1.13(b) shows the result of the level-set models that uses 1.13(a) as an initial state and has a value of β equal to 0.5. The result is a reasonable reconstruction of the noiseless model (Figure 1.13(c)) which combines the six points of view and the smoothing function.

Figure 1.14(a) shows a range map taken with the Perceptron model P5000, an infra-red, time-of-flight laser range finder with a pan-tilt mechanism. Figure 1.14(b) shows the amplitudes associated with the return signal (an *intensity*), and 1.14(c) shows a surface plot of the range map to demonstrate the degree of noise (additive and outliers). Figure 1.14(d) shows the confidence values associated with those range measurements. These confidence values are derived from empirical data about the level of noise in the range finder (which depends on the return amplitude), and some analysis, from first principles, about the effects of uncertainty in the 3D positions of the scans and the model — which results in the lower confidence at edges as described in [33]. We combined twelve such views from different locations in the room to generate the results that follow.

Figure 1.15(a) shows the initial estimate based on the zero crossings of $g(\boldsymbol{x})$, and 1.15(b) shows the result of 32 iterations with the prior term and the correction for the surface normal direction. The size of the volume is 300 × 150 × 180 voxels, and the resolution is 1.8 cm/voxel. These results show the ability of the statistically-based approach to overcome the noise in the scanner, and they show that the inclusion of iterative, model-fitting scheme helps create more accurate reconstructions. The resolution of the model falls below that of the scans, because it was limited by the random-access-memory available on our workstation.

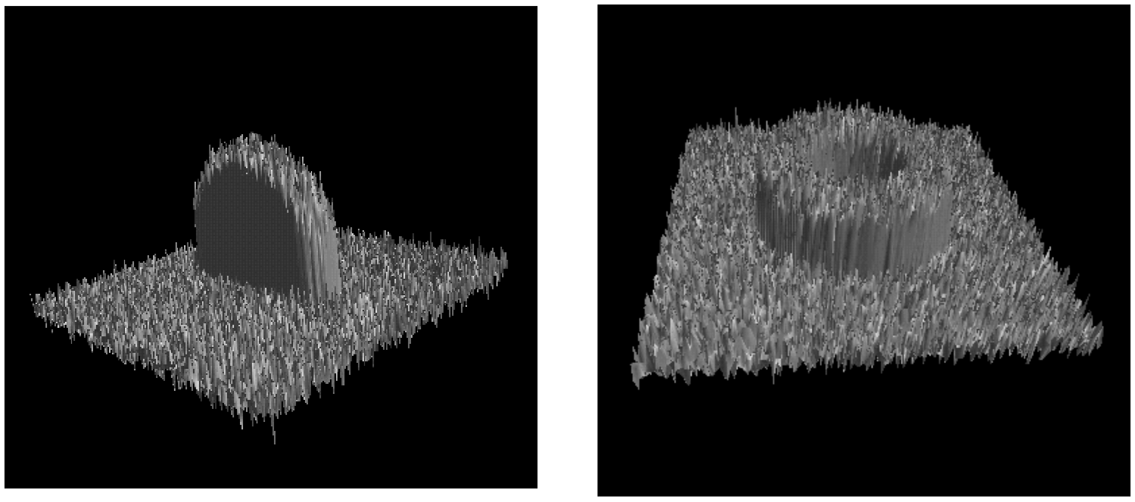


Figure 1.12: Range maps: Synthetic range data 200×200 pixels with 20% Gaussian white noise of a torus end (a) and side (b).

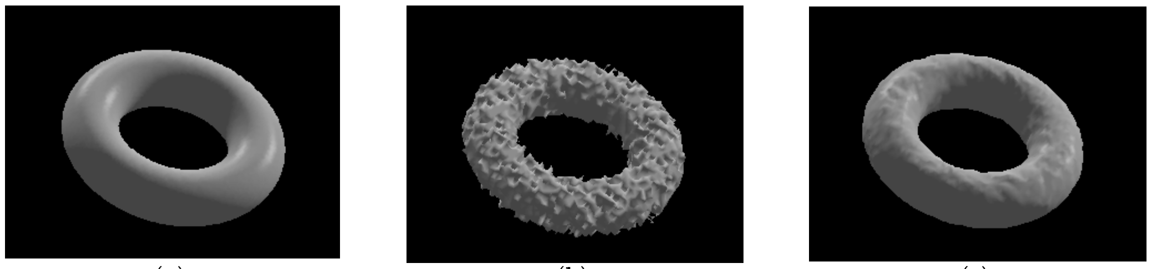


Figure 1.13: (a) An analytically-defined model of a torus. (b) An initial model (80×80×40 voxels) is constructed by combining six points of view of a torus and solving for $g(\boldsymbol{x}) = 0$. (c) The model, which is attracted to the range data but subject to internal forces, evolves and settles into a smoother steady state.

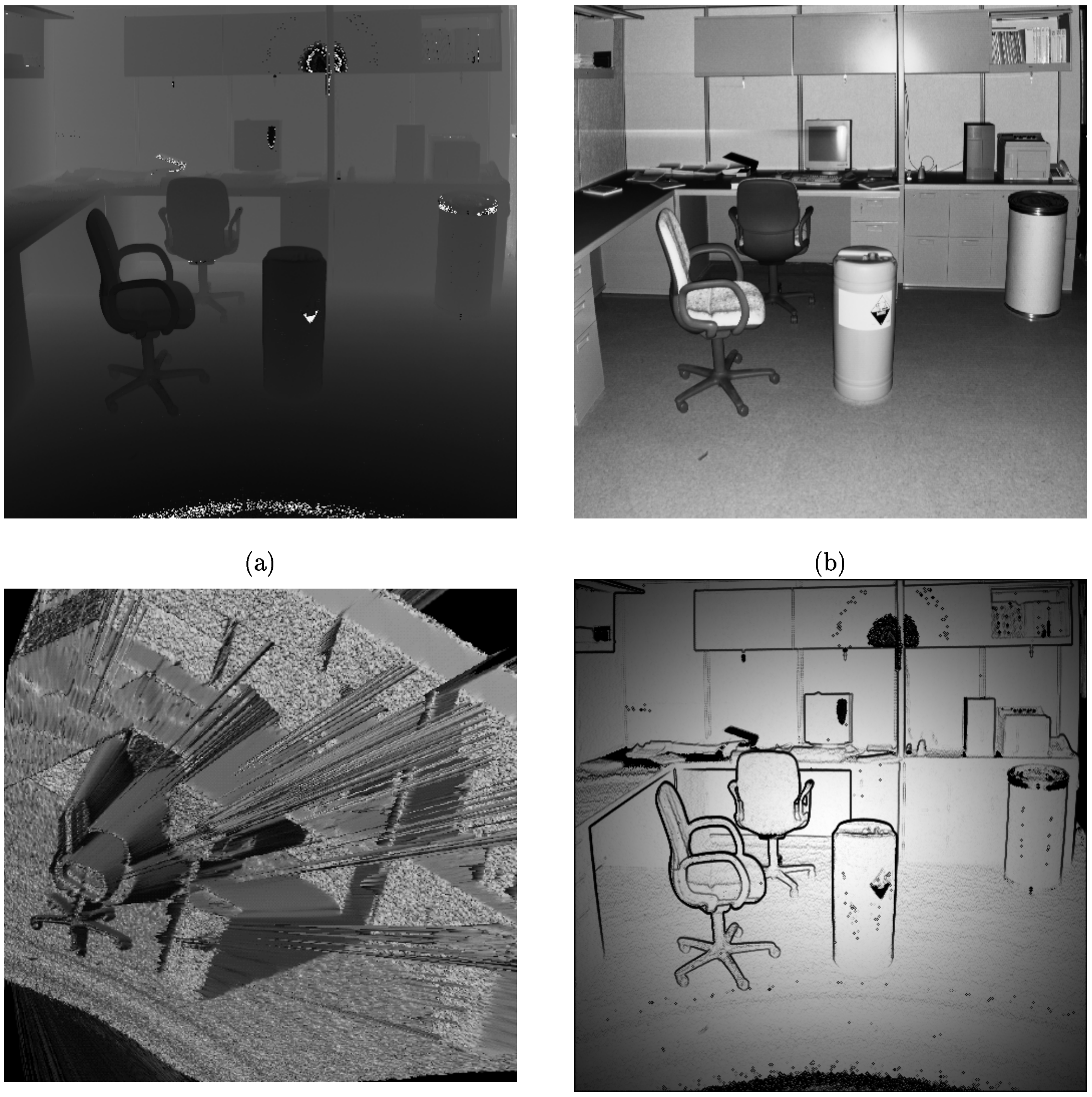
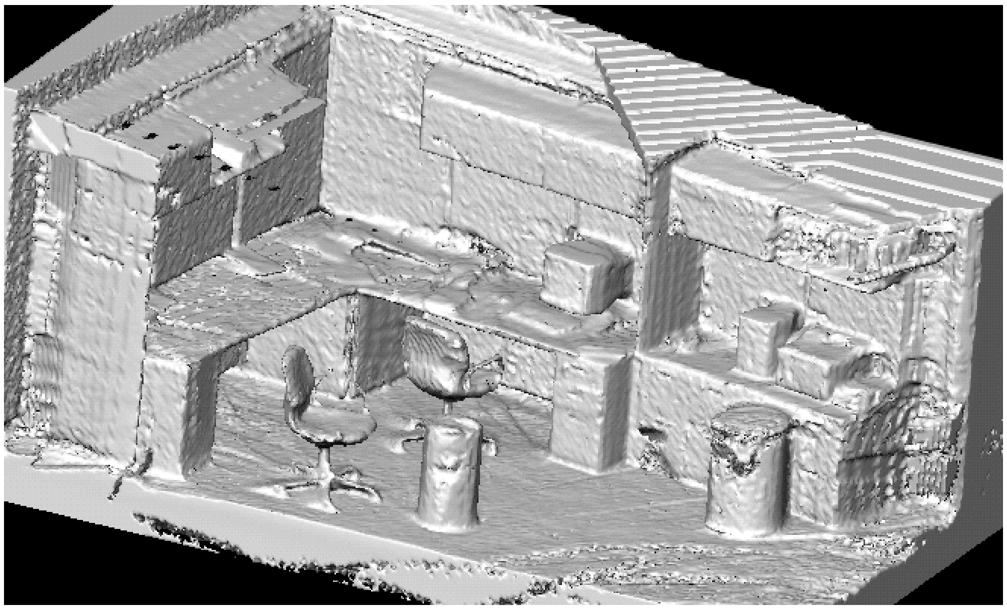

(a) (b) (c) (d)

Figure 1.14: (a) One of twelve range maps (b) The associated amplitude map (c) A surface plot of the range data to show the level of noise. (d) The confidence measures associated with those range values.



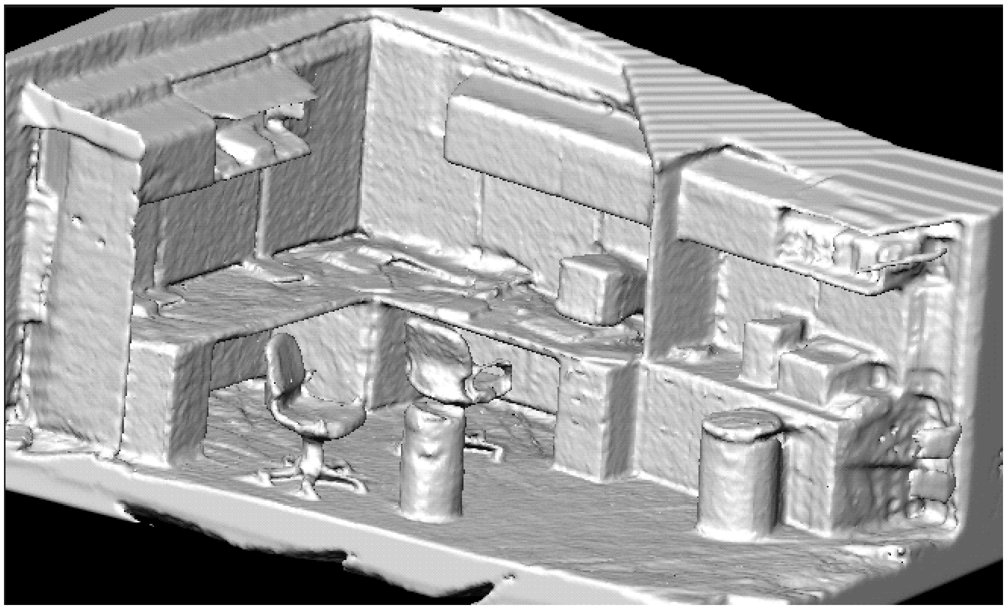


Figure 1.15: (top) The 3D reconstruction resulting from the zero crossings of $g(\boldsymbol{x})$ gives some averaging, but includes no prior. (bottom) The result of 32 iterations with the iterative scheme includes the prior and excludes influences of data on surfaces that face away from the scanner.

Some small features, such as the arm rests of the chairs, are lost because of the inaccuracies in the registration of the individual range maps.

1.8 VISPACK

1.8.1 Introduction

VISPACK is a set of *C++*, object-oriented libraries for image processing, volume processing, and level-set surface modeling. It consists of five libraries: Matrix, Image, Volume, Util, and Voxmodel (level-set modeling). These libraries can be used separately or together when creating applications.

VISPACK incorporates eight basic design attributes. These are

Data Handles/Copy on Write: VISPack is an object-oriented library, and as such we allow the objects to handle memory management, and relieve the programmer (in most cases) from having to worry pointers and the corresponding memory allocation/deallocation problems. For this we use the data handles with a *copy on write protocol.* Copy constructors perform a shallow copy with reference counting until a *non const* operation on the underlying buffers forces a deep copy. Thus deep copies are performed only when necessary, but all memory is maintained by the objects and objects behave as "variables" rather than pointers.

Modified Data Hiding: Access to data in objects is generally through access methods, however, pointers to buffers for fast implementations are available.

Templates: VISPack utilizes the templating construct of C++ virtually throughout. Many of the objects, including images, volumes, lists, and arrays, are intended to support a wide range of data types. Thus, via templating programmers can define the pixels of different images of different types, such as floating point, 24-bit color, and 16-bit greyscale.

Use of Standard File Formats: When appropriate VISPack uses standard file formats. We choose formats that are well known and have publicly available libraries that can be distributed with our libraries. The matrix library uses a simple text format. The image library uses TIFF and FITS file formats. Because no standard format exists for saving volumes of data we do use a *raw* file format.

Operator Overloading: Proper use of operator overloading gives users a convenient way to execute operations on an object. When compined with the copy-on-write convention, operator overloading allows programmers to treat many heavy-weight objects (e.g. images and volumes) as variables. For instance, the following code computes non-maximal edges in a on a filtered volume.

```
Volume<float> dx, dy, dz;
Volume<float> vol_gauss = vol.gauss(0.5);
Volume<float> vol_out = (((dx = vol_gauss.dx()).power(2) *vol_gauss.dx(2)
       + ((dy = vol_gauss.dy()).power(2)*vol_gauss.dy(2)
       + ((dz = vol_gauss.dz()).power(2)*vol_gauss.dz(2)
       + dx*dy*(dx).dy() + dx*dz*(dx).dz())
       + dy*dz*(dy).dz()) )).zeroCrossings() && ((dx.power(2) + dy.power(2)) > T*T));
```

1.8.2 Level-Set Surface-Modeling Library

The Level-Set Surface-Modeling (LSSM) Library is an implementation of the level-set technique [7, 8] specifically for deforming surface models embedded in volumes. The implementation uses the sparse-field method described in [17]. The library implements all of the basic numerical algorithms and handles all of the data structures required to perform LSSM. The strategy for using this library is to subclass the object `VoxModel`, set some parameters, define a set of simple virtual functions that control the deformation process, initialize the model, and then direct the model to iteratively deform according to those equations. This section describes the relationship between the mathematics of previous sections and the VISPack library. Its also presents an example of using VISPack libarary to do 3D shape metamorphosis as described in Section 1.7.1.

Figure 1.16: The base class `LevelSetModel` keeps track of the active set and surrounding layers, while the derived class `Voxmodel` performs updates on the active set from a set of virtual functions.

Surface Deformation

The LSSM library allows one to solve for surface deformations, as a function of time, for general level-set surface movements of the form:

$$\frac{\partial \boldsymbol{x}}{\partial t} = \alpha \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{N}(\boldsymbol{x})) + \beta G(\boldsymbol{x}, \boldsymbol{N}(\boldsymbol{x}))\boldsymbol{N}(\boldsymbol{x}) + \gamma \boldsymbol{N}(\boldsymbol{x}) + \eta E\left(H(\boldsymbol{x}), K(\boldsymbol{x})\right), \tag{1.40}$$

where $\boldsymbol{x}$ is a point on the surface. This equation is solved by representing the surface as the kth level set of an implicit function $\phi(\boldsymbol{x}, t) : \mathrm{I\!R}^3 \times \mathrm{I\!R}^+ \mapsto \mathrm{I\!R}$. This gives

$$\frac{\partial \phi}{\partial t} = \alpha \boldsymbol{F}(\boldsymbol{x}, \nabla\phi)) \cdot \nabla\phi + \beta G(\boldsymbol{x}, \nabla\phi)|\nabla\phi| + \gamma|\nabla\phi| + \eta E(D\phi, \mathrm{D}^2\phi), \tag{1.41}$$

where $\mathrm{D}\phi$ and $\mathrm{D}^2\phi$ are collections first and second derivatives of ϕ, respectively. This equation is solved on a discrete grid using an *up-wind* scheme gradient calculations, centralized differences for the curvature, and forward finite differences in time. The LSSM library uses the *sparse-field* method described in Section 1.6.3 and in [18].

Thus, the LSSM library offers the following capabilities:

1. Creates an initial model (with associated active set) from a volume.
2. Calculates $\Delta u^n_{i,j,k}$ and Δt using virtual functions (defined by subclasses) that describe $\boldsymbol{F}$ and G, and parameters (values set by the subclass) α, β, γ, and η.
3. Performs an update on the values of $u^n_{i,j,k}$.
4. Maintains the list of active grid points and updates the *layers* around those points in order to maintain a neighborhood from which to calculate subsequent updates.
5. Provides access to the volume that defines $u^n_{i,j,k}$ and the linked list of active grid points.

Given the volume defining $u^n_{i,j,k}$, one can then rely on the functionality of the volume library for subsequent processing, file I/O, or surface extraction.

Structure and Philosophy of the LSSM Library

The library is organized (mostly for ease of development) into a base class, `LevelSetModel`, and a derived class, `VoxModel`, as shown in Figure 1.16. The base class does all of the book keeping associated with the active set and surrounding *layers*, the link lists associated with those sets, and initializing the model. Thus it adds and removes voxels from the active set (and surrounding layers) in response to an update operation. The base class assumes that the subclasses know how to update individual voxels.

The subclass, `VoxModel`, performs update on the grid points in the active set of the form given in Equation 1.16, using functions $\boldsymbol{F}$ and G and parameters α, β, γ, and η. It also calculates the maximum Δt that ensures stability. Thus a user who wishes to perform a surface deformation using the LSSM library, would create subclass of `VoxModel` and define the appropriate virtual functions and set the parameters to achieve the desired behavior.

The `LevelSetModel` Object

The `LevelSetModel` contains a volume of values, a volume of status flags, five lists (one active list, two inside lists, and two outside lists), and three parameters that determine the origin of the coordinate system form which the model performs its calculations.

There are two constructors, `LevelSetModel()` and `LevelSetModel( const VISVolume<float> &)`. The first simply initializes the data structure, and the second also set the values of the model volume (`_values`) to the input. Once the values have been set, one can create an initial volume from those values by calling `constructLists()`, which can also take a floating-point argument that controls the scaling of the input relative to a local distance transform near the zero set.

The list that keeps track of the active set, called `_active_list`, keeps track of the location of those grid points and a single floating-point value, which stores the change in their values from one iteration to the next.

Another important methods for users of this object is `update(float)`, which changes the grey-scale values of the grid for the active set according to the values stored in `_active_list`, and updates the status of elements on the active list as well as the values and status of nearby layers (2 inside and 2 outside). The floating point argument is the value of Δt from Equation 1.16, and the return value is the maximum change that occurred on the active set. Finally, the method `iterate()` calls the virtual method `calculate_change`, a virtual function which sets the values of $\Delta u^n_{i,j,k}$ and returns the maximum value of Δt for stability, and then calls `update`. For this object the function `calculate_change` performs some trivial (i.e., useless) operation.

The `VoxModel` Object

The `VoxModel` object is a subclass of `LevelSetModel`, and it add three things to the base class.

1. `calculate_change()` is redefined to implement the surface deformation described in Equation 1.41.
2. The virtual functions are declared for F (called `force`) and G (called `grow`). These functions are defined to return zero for this object.
3. The parameters that control the relative influence of the various terms are read from file by a routine `load_params`.
4. A method `rescale(float)` is defined, which resamples the volume of grid-point values into a new volume with different resolution and redefines the lists (and thereby the model) in this new volume. This method is for performing coarse-to-fine deformation procedures.

1.8.3 Example: 3D Shape Metamorphasis

The `Morph` object allows one to construct a sequence of volumes or surface meshes using the 3D shape metamorphasis technique described in Section 1.7.1, which was first proposed by Whitaker and Breen [17]. This technique relies distance transforms for both the source and target objects and uses a LSSMs to manipulate the shape of the source so that it coincides with the target. The surface deformation that describes this behavior is

$$\frac{\partial \boldsymbol{x}}{\partial t} = \beta G\left(T(\boldsymbol{x})\right) \boldsymbol{N}(\boldsymbol{x}), \tag{1.42}$$

where $G(\boldsymbol{x})$ is simply the distance transform (or some monotonic function thereof) of the target, and T is a coordinate transformation that aligns the source and target objects. The level-set formulation of this is

$$\frac{\partial \phi(\boldsymbol{x}, t)}{\partial t} = \beta G\left(T(\boldsymbol{x})\right) |\nabla \phi|. \tag{1.43}$$

The morphing process consists of several steps:

1. Read in distance transforms (in the form of volumes) for both source and target.
2. Initialize the LSSM by fitting it to the zero set of the source distance transform.
3. Update the LSSM according to Equation 1.43.
4. Save intermediate volumes/surfaces at regular intervals.

The remainder of this section lists the code and comments for three files, morph.h (which declares the `Morph` object), morph.C (which defines the methods) and main.C (which performs all of the I/O and uses the `Morph` object to construct a sequence of shapes.

1.8.4 Morph.h

```
//
// morph.h
//
//

#ifndef iris_morph_h
#define iris_morph_h

#include "voxmodel/voxmodel.h"
#include "matrix/matrix.h"

#define INIT_STATE 0
#define MORPH_STATE 1
//
// This is the morph object.  It uses all of the machinery of the base
// class to manipulate level sets.  It needs to have an initial volume
// and a final volume (which would typically be the distance transform,
// it might need a 3D transformation, and it needs to redefine the
// virtual function "grow", which takes 6 floats as input, the position
// followed by the normal vectors (all will calculated and passed into
// this method by the base class).  It might also have a state, that
// indicates whether or not it's been initialized.
//
// Functions not defined here should be defined in "morph.C"
//
class Morph: public VoxModel
{
  protected:
    VISVolume<float> _dist_source;
    VISVolume<float> _dist_target;
    VISMatrix _transform;
//
// This is the function that is used by the base class to manipulate the
level
// set.  You can define it to by anything you want.  For this object, it
will
// return a value from the distance transform of the target.
//
    virtual float grow(float x, float y, float z,
          float nx, float ny, float nz);

// There are two states.  In the first state, the model is trying to fit
// to the input data.  In this way the models starts by looking just like

// the input data
    int _state;

  public:

    Morph(const Morph& other)
    {
 _dist_target = other._dist_target;
 _initial = other._initial;
 _state = MORPH_STATE;
 _transform = VISVISMatrix(3, 3);
 _transform.identity();
// initialize();
    }

    Morph(VISVolume<float> init, VISVolume<float> d)
 :VoxModel()
    {
 _dist_target = d;
 _initial = init;
 _state = MORPH_STATE;
 _transform = VISVISMatrix(3, 3);
 _transform.identity();
// initialize();
    }

    void initialize();

// for this object I assume that the transform is just a matrix.
// but it could be anything
    void transform(const VISVISMatrix& t)
    { _transform = t;}

    const VISVISMatrix& transform()
    { return(_transform);}

    void distance(const VISVolume<float> d)
    { _dist_target = d;}
    VISVolume<float> distance()
    { return(_dist_target);}

};
#endif
```

1.8.5 Morph.C

```
#include "morph.h"
#include "util/geometry.h"
#include "util/mathutil.h"


//
// this is the virtual function, that  is the guts of it all.
//

float Morph::grow(float x, float y, float z,
    float nx, float ny, float nz)
{

// this says you are in the morph state (things have been initialized)
    if (_state == MORPH_STATE)
```

```
 {
 float xx, yy, zz;
 VISPoint p(4u);
 p.at(0) = x;
 p.at(1) = y;
 p.at(2) = z;
 p.at(3) = 1;
 VISPoint p_tmp;
// this is where you could put some other transform.
 p_tmp = _transform*p;

 xx = p_tmp.x();
 yy = p_tmp.y();
 zz = p_tmp.z();

// make sure you are not out of the bounds
// of your distance volume.
 if (_dist_target.checkBounds(xx, yy, zz))
// if not, get the distance (use trilinear interpolation).
     return(_dist_target.interp(xx, yy, zz));
 else
     return(0.0f);
    }
else
    {
// if you are still initializing, then move toward the zero set of
// your initial case
 if (_initial.checkBounds(x, y, z))
     return(_initial.interp(x, y, z));
 else
     return(0.0f);
    }
}

// this makes the model look like the input.
#define INIT_ITERATIONS 5
void Morph::initialize()
{
    _values = _initial;
    int state_tmp = _state;
    _state = INIT_STATE;
    construct_lists(DIFFERENCE_FACTOR);
// these couple of iterations are required to make sure that the zero
// sets of the model match the zero sets of the
//
    for (int i = 0; i < INIT_ITERATIONS; i++)
 {
// limit the dt to 1.0 so that the model settles in to a solution
     update(::min(calculate_change(), 1.0f));
 }
    _state = state_tmp;
}
```

1.8.6 Main.C

```
#include "vol/volume.h"
#include "vol/volumefile.h"
#include "image/imagefile.h"
#include "morph.h"
#include <string.h>


const int V_HEIGHT = (40);
const int V_WIDTH = (40);
const int V_DEPTH = (40);

#define XY_RADIUS (12)  // this matches the 2.5D data generated in
torus.C
#define T_RADIUS (4)  // this matches the 2.5D data generated in torus.C
#define S_RADIUS (12)  // radius of a sphere

#define B_WIDTH (20.0f)
#define B_HEIGHT (60.0f)
#define B_DEPTH (20.0f)

#define B_CENTER_X (12.0f)
#define B_CENTER_Y (32.0f)
#define B_CENTER_Z (12.0f)

float sphere(unsigned x, unsigned y, unsigned z);
float torus(unsigned x, unsigned y, unsigned z);
float cube(unsigned x, unsigned y, unsigned z);


// This is a program that does the morph.  If you give it two
// arguments, it reads the initial model and the dist trans for the
// final model from the two file names given, otherwise, it makes a
sphere
// and deforms it into a torus

main(int argc, char** argv)
{

VISVolume<float> vol_source, vol_target;
VISVolumeFile vol_file;
int i;
char fname[80];

vol_source = VISVolume<float>(25,65,25);
vol_source.evaluate(cube);

if (argc > 2)
    {
// read in the sourceing model
 vol_source = VISVolume<float>(vol_file.read_float(argv[1]));
// read in the dist trans of the final model
     vol_target = VISVolume<float>(vol_file.read_float(argv[2]));
    }
else
// make up some volumes
    {
 vol_source = VISVolume<float>(V_WIDTH, V_HEIGHT, V_DEPTH);
 vol_source.evaluate(sphere);
 vol_target = VISVolume<float>(V_WIDTH, V_HEIGHT, V_DEPTH);
 vol_target.evaluate(torus);
    }

// create morph object
Morph morph(vol_source, vol_target);
// loads in some parameters (for morphing these are all zero but one)
// i.e.
//
//
//
//
morph.load_parameters("morph_params");
morph.initialize();
vol_file.write_float(morph.values(), "morph0.flt");

float dt;

// do 150 iterations for your model to get from start to finish
// probably don't need this many iterations

for (i = 0; i < 150; i++)
    {
 dt = morph.calculate_change();
// limit dt to 0.5 so that model never overshoots goal
 dt = min(dt, 0.5f);
 morph.update(dt);

 printf("iteration %d dt %f\n", i, dt);

 if (((i + 1)%10) == 0)
 {
// save every tenth volume
     sprintf(fname, "morph_out.%d.dat", i + 1);
     vol_file.write_float(morph.values(), fname);
 }
    }

// save a surface model (i.e. marching cubes).
vol_file.march(0.0f, morph.values(), ``morph_final.iv'');

printf("done\n");

}
```

Bibliography

[1] J. Blinn, "A generalization of algebraic surface drawing," *ACM Trans. on Graphics*, vol. 1, pp. 235–256, March 1982.

[2] S. Muraki, "Volumetric shape description of range data using "blobby model"," in *SIGGRAPH '91 Proceedings* (T. W. Sederberg, ed.), pp. 227–235, July 1991.

[3] W. Lorenson and H. Cline, "Marching cubes: A high resolution 3D surface construction algorithm," *Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1982.

[4] J. Sethian, *Level Set Methods and Fast Marching Methods.* Cambridge: Cambridge University Press, second ed., 1999.

[5] M. Kass, A. Witkin, and D. Terzopoulos, "Snakes: Active contour models," *International Journal of Computer Vision*, vol. 1, pp. 321–323, 1987.

[6] D. Terzopoulos and K. Fleischer, "Deformable models," *The Visual Computer*, vol. 4, pp. 306–331, December 1988.

[7] S. Osher and J. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations," *Jrnl. of Comp. Phys.*, vol. 79, pp. 12–49, 1988.

[8] J. A. Sethian, *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision, and Material Sciences.* Cambridge University Press, 1996.

[9] L. Alvarez and J.-M. Morel, "A morphological approach to multiscale analysis: From principles to equations," in *Geometry-Driven Diffusion in Computer Vision* (B. M. ter Haar Romeny, ed.), pp. 4–21, Kluwer Academic Publishers, 1994.

[10] V. Caselles, R. Kimmel, and G. Sapiro, "Geodesic active contours," in *Fifth Int. Conf. on Comp. Vision*, pp. 694–699, IEEE, IEEE Computer Society Press, 1995.

[11] B. B. Kimia and S. W. Zucker, "Exploring the shape manifold: the role of conservation laws.," in *Shape in Picture: the mathematical description of shape in greylevel images* (Y.-L. O, A. Toet, H. Heijmans, D. H. Foster, and P. Meer, eds.), Springer-Verlag, 1992.

[12] R. Malladi, J. A. Sethian, and B. C. Vemuri, "Shape modeling with front propagation: A level set approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 2, pp. 158–175, 1995.

[13] R. T. Whitaker and D. T. Chen, "Embedded active surfaces for volume visualization," in *SPIE Medical Imaging 1994*, (Newport Beach, California), 1994.

[14] S. Kichenassamy, A. Kumar, P. Olver, A. Tannenbaum, and A. Yezzi, "Gradient flows and geometric active contour models," in *Fifth Int. Conf. on Comp. Vision*, pp. 810–815, IEEE, IEEE Computer Society Press, 1995.

[15] A. Yezzi, S. Kichenassamy, A. Kumar, P. Olver, and A. Tannenbaum, "A geometric snake model for segmentation of medical imagery," *IEEE Transactions on Medical Imaging*, vol. 16, pp. 199–209, April 1997.

[16] L. Lorigo, O. Faugeraus, W. Grimson, R. Keriven, and R. Kikinis, "Segmentation of bone in clinical knee MRI using texture-based geodesic active contours," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI '98)* (W. Wells, A. Colchester, and S. Delp, eds.), pp. 1195–1204, October 1998.

[17] R. Whitaker and D. Breen, "Level-set models for the deformation of solid objects," in *The Third International Workshop on Implicit Surfaces*, pp. 19–35, Eurographics, 1998.

[18] R. T. Whitaker, "A level-set approach to 3D reconstruction from range data," *Int. Jrnl. of Comp. Vision*, vol. October, no. 3, pp. 203–231, 1998.

[19] S. Osher and J. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations," *Jrnl. of Comp. Phys.*, vol. 79, pp. 12–49, 1988.

[20] D. Adalstein and J. A. Sethian, "A fast level set method for propagating interfaces," *Jrnl. of Comp. Phys.*, pp. 269–277, 1995.

[21] A. Lerios, C. D. Garfinkle, and M. Levoy, "Feature-Based volume metamorphosis," in *SIGGRAPH '95 Conference Proceedings* (R. Cook, ed.), Annual Conference Series, pp. 449–456, Addison Wesley, Aug. 1995.

[22] J. Kent, W. Carlson, and R. Parent, "Shape transformation for polyhedral objects," in *SIGGRAPH '92 Proceedings*, pp. 47–54, July 1992.

[23] J. Rossignac and A. Kaul, "AGRELS and BIPs: Metamorphosis as a bezier curve in the space of polyhedra," *Computer Graphics Forum (Eurographics '94 Proceedings)*, vol. 13, pp. C–179–C–184, September 1994.

[24] J. F. Hughes, "Scheduled Fourier volume morphing," in *Computer Graphics (SIGGRAPH '92 Proceedings)* (E. Catmull, ed.), vol. 26, pp. 43–46, July 1992.

[25] B. Payne and A. Toga, "Distance field manipulation of surface models," *IEEE Computer Graphics and Applications*, vol. 12, no. 1, pp. 65–71, 1992.

[26] D. Cohen-Or, D. Levin, and A. Solomivici, "Three-dimensional distance field metamorphosis," *ACM Transactions on Graphics*, vol. 17, no. 2, pp. 116–141, 1998.

[27] P. Getto and D. Breen, "An object-oriented architecture for a computer animation system," *The Visual Computer*, vol. 6, pp. 79–92, March 1990.

[28] D. Breen, S. Mauch, and R. Whitaker, "3D scan conversion of CSG models into distance volumes," in *Proceedings of the 1998 Symposium on Volume Visualization*, pp. 7–14, ACM SIGGRAPH, October 1998.

[29] A. Middleditch and K. Sears, "Blend surfaces for set theoretic volume modeling systems," in *SIGGRAPH '85 Proceedings*, pp. 161–170, July 1985.

[30] J. Bloomenthal and K. Shoemake, "Convolution surfaces," in *SIGGRAPH '91 Proceedings* (T. W. Sederberg, ed.), pp. 251–257, July 1991.

[31] M. Desbrun and M.-P. Gascuel, "Animating soft substances with implicit surfaces," in *SIGGRAPH '95 Proceedings*, pp. 287–290, August 1995.

[32] R. T. Whitaker, "Volumetric deformable models: Active blobs," in *Visualization In Biomedical Computing 1994* (R. A. Robb, ed.), (Mayo Clinic, Rochester, Minnesota), pp. 122–134, SPIE, 1994.

[33] R. T. Whitaker, "A level-set approach to 3D reconstruction from range data," *Int. Jrnl. of Comp. Vision*, vol. October, no. 3, pp. 203–231, 1998.

[34] G. Turk and M. Levoy, "Zippered polygon meshes from range images," in *Proc. of SIGGRAPH '94*, pp. 311–318, ACM SIGGRAPH, August 1994.

[35] Y. Chen and G. Médioni, "Fitting a surface to 3-D points using an inflating ballon model," in *Second CAD-Based Vision Workshop* (A. Kak and K. Ikeuchi, eds.), vol. 13, pp. 266–273, IEEE, 1994.

[36] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proc. of SIGGRAPH '96*, pp. 303–312, ACM SIGGRAPH, August 1996.

[37] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, "Surface reconstruction from unorganized points," *Computer Graphics*, vol. 26, no. 2, pp. 71–78, 1992.

[38] A. Hilton, A. J. Stoddart, J. Illingworth, and T. Windeatt, "Reliable surface reconstruction from multiple range images," in *Euro. Conf. on Comp. Vision*, Springer-Verlag, 1996.

[39] C. Bajaj, F. Bernardini, and G. Xu, "Automatic reconstruction of surfaces and scalar fields from 3D scans," in *SIGGRAPH '95 Proceedings*, pp. 109–118, August 1995.